Naval Research Laboratory

Stennis Space Center, MS 39529-5004

NRL/MR/7320--13-9441

Gerris Flow Solver: Implementation and Application

Timothy R. Keen
Timothy J. Campbell
James D. Dykes
Paul J. Martin

Ocean Dynamics and Prediction Branch
Oceanography Division

May 12, 2013

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
12-05-2013	Memorandum Report	

4. TITLE AND SUBTITLE

Gerris Flow Solver: Implementation and Application

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
0601153N

6. AUTHOR(S)

Timothy R. Keen, Timothy J. Campbell, James D. Dykes, and Paul J. Martin

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
73-4261-02-5

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Oceanography Division
Stennis Space Center, MS 39529-5004

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/7320--13-9441

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
One Liberty Center
875 North Randolph Street, Suite 1425
Arlington, VA 22203-1995

10. SPONSOR / MONITOR'S ACRONYM(S)

ONR

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes the implementation of the Gerris Flow Solver (GFS) at NRL. GFS is an open-source software library for computational fluid dynamics that includes modules for the solution of the Navier-Stokes equations, and both linear and non-linear shallow water equations. GFS features dynamic adaptive mesh refinement (AMR) based on a semi-structured quadtree/octree mesh. A visualization tool (GfsView) is included that enables viewing model output on the adaptive mesh. We have applied GFS to a number of problems in estuary flow. The CFD model has been used for direct numerical simulation of 2D-vertical tidal flow in a macrotidal river, and interaction of a fluid mud layer with the flow. The 2D shallow water models (linear and non-linear) have been used to simulate tidal flow in Mississippi Sound and the Gulf of Maine/Bay of Fundy. We have implemented a wave-current bottom boundary layer model (BBLM) into GFS as a plug-in.

15. SUBJECT TERMS

Gerris Flow Solver
Adaptive mesh refinement
Finite volume
Numerical model

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Timothy Keen
Unclassified Unlimited	Unclassified Unlimited	Unclassified Unlimited	Unclassified Unlimited	189	19b. TELEPHONE NUMBER *(include area code)* (228) 688-4950

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

Executive Summary E-1
Section 1: Project Overview 1
Introduction 1
Background 1
Objectives 2
Cited References 2
Publications and Presentations 4
Document Organization 4
Section 2: Implementation 6
Local Build Information 6
Creating a User-Specific Build 6
Settings for Run Environment 7
Local Batch System 8
A Demo 8
Package Dependencies for GTS, Gerris, & GfsView 8
Details on Package Dependencies 9
Section 3: Gerris Flow Solver Plug-Ins 16
Introduction 16
GfsFunction 16
Read a Function from the Simulation File 17
Compose and Compile a GfsFunction 18
Create a GModule as a Plug-in 18
GfsFunctionConstant Example 18
GfsFunction Example with GfsVariables 19
GfsModule 21
Section 4: Gerris Model Domains 23
Defining a Domain with a GTS File 23
Creating a GTS Domain File 24
Sensitivity Testing for GTS Domains 26
Terrain Databases (KDT) 28
Local KDT Databases 29

Using The Terrain Module 32
Terrain Data Base Example 32
Section 5: Boundary Conditions 39
Introduction 39
Tidal Boundary 40
Constant Values Supplied in Simulation File 41
Tides Module (from FES2004 only) 43
Extracting tides from a database 44
Tidal Constituents from GTS Files 45
Flather Boundary Condition 46
Surface Forcing with Wind 51
Section 6: Gerris Input/Output Processing and GIS 53
GNU Triangulated Surface (GTS) Files 53
Output Arc Grid File 54
Map Projections 54
Using the MapProjection Module 54
Projecting Arc Grid files in ArcMAP 55
Reformatting ArcInfo Grid Files to NetCDF 55
Reformatting to COARDS without Georeferencing 56
Creating Georeferenced NetCDF Files 59
Creating a COARDS-compatible NetCDF File 66
Section 7: Testing CFD Solvers 68
Two-Dimensional CFD Testing 68
Dimensionless Scaling 69
Oscillatory flow 71
Logarithmic current profile 72
Volume of Fluid (VOF) Simulation 76
Section 8: Model Setup 80
Using the *GfsOcean* Module 80
Introduction 80
Domain Definition with GTS Files 80
Tidal Boundary Condition 80

Surface Forcing with Wind 80
Using the *GfsRiver* Module 80
Section 9: Lock Exchange Simulations 81
Introduction 81
Background 82
Simulations with a Non-Hydrostatic 2D Model 85
Results 86
Discussion 87
Simulations with Gerris (2D CFD) 89
Hartel et al. (1997) 90
Hartel et al. (2000) 92
Maxworthy et al. (2002) 96
O'Callaghan et al. (2010) 98
Summary 102
References Cited 102
Section 10: Example Applications 104
Tidal Simulation in the Gulf of Maine 104
Introduction 104
Objectives 106
Methods 107
Results 116
Summary 120
References 120
Non-Acoustic Optical Vulnerability Assessment Software (NOVAS) 121
Background 121
Objectives 122
General approach 123
Modeling approach with Gerris 127
Results 128
References Cited 135
Gerris Ice Dynamics 136
Introduction 136

Method 136
Results 137
References 140
Other applications 140
Appendices 141
Appendix A. Model Structure and Operation 141
Introduction 141
The GfsSimulation Class 141
The GfsDomain Class 143
The GfsInit Class 144
GfsSurfaceClass 145
Bathymetry as a GfsSolid 148
Tidal Constituents as Gfsinit (GfsEvent) Objects 149
Overview of a simulation 152
References 163
Appendix B. GNU Triangulated Surface (GTS) Library 164
Introduction 164
GTS Objects, Classes, Constructors, and Inheritance 165
GtsSurfaceClass 169
GtsGraph Class 177
GtsContainers and GtsContainees 180
GtsHashTables 183

Executive Summary

This report describes the Gerris Flow Solver (GFS), which is also known as Gerris. GFS was developed primarily by Stephane Popinet of the National Institute of Water and Atmospheric Research (NIWA) of New Zealand.

The Gerris Flow Solver (GFS) -- a free, open source, software system for computational fluid dynamics -- includes modules for the solution of time-dependent incompressible variable-density Euler, Stokes or Navier-Stokes equations, and for the solution of both linear and non-linear shallow water equations. GFS is designed as a reusable, object-oriented library of functions that facilitate the implementation of new models. GFS features dynamic adaptive mesh refinement (AMR) based on a semi-structured quadtree/octree mesh. A visualization tool (GfsView) is included that enables viewing model output on the adaptive mesh. The latest release of GFS is maintained in /common/gfs and is available for general use. We have applied GFS to a number of problems in estuary flow. The CFD model has been used for direct numerical simulation of 2D-vertical tidal flow in a macrotidal river, and interaction of a fluid mud layer with the flow. The 2D shallow water models (linear and non-linear) have been used to simulate tidal flow in Mississippi Sound and the Gulf of Maine/Bay of Fundy. In this talk we will give an overview of GFS (implementation and usage) along with a discussion of results from several application areas.

Section 1: Project Overview

Introduction

Ocean modeling has progressed rapidly in the last 20 years. The 1960's and 1970's saw the development of finite-difference solutions of the Reynolds Averaged Navier-Stokes (RANS) equations on Cartesian grids with no vertical layers (e.g., Jelesnianski, 1966) or fixed (z) levels for vertical discretization (Leendertse et al., 1973). As today, these models were applied to storm surge prediction and contaminant transport in estuaries, respectively. One of the most successful numerical ocean models is the Princeton Ocean Model (POM) (Blumberg and Mellor, 1987), which is still widely used. This model remains as a standard RANS formulation used in ocean prediction today (Yin et al., 2010), with terrain-following vertical coordinates and curvilinear horizontal coordinates. The finite-difference method was known by Euler in ~1768 and was applied in its modern form in the 1950's. Thus, modern electronic computers have allowed the effective solution of nineteenth century equations. The difficulty of accurately simulating the multi-scale nature of physical processes in the ocean has proven problematic, however, and curvilinear coordinate systems and nested grids have been used to resolve these scales.

The multi-scale problem in ocean flows was addressed in part by application of the finite element method (FEM) that was formalized by Zienkiewicz of the Imperial College (e.g., Zienkiewicz, 1966). It is the solver for the Imperial College Ocean Model (ICOM), which uses 3D adaptive mesh methods (Ford et al., 2004). The finite volume method (FVM) was developed in the 1990's (Billett and Toro, 1996).

Background

Adaptive Mesh Refinement (AMR) was used by Berger and Jameson (1985) for a finite volume solution of the steady Euler equations on an airfoil. They used the error in the solution to increase local grid resolution. They used rectangles (2D) to standardize the solution and simplify the adaptation process. They used multiple levels of refinement. Each component grid has its own solution vector. The boundary conditions are supplied for each grid without interpolation in order to conserve energy and mass.

Popinet and Zaleski (1999) describe a front-tracking algorithm for the 2D incompressible Navier-Stokes (N-S) equations that uses a staggered marker and cell (MAC) method for the pressure, volume fraction, momentum, and velocity discretization on a Cartesian grid. The solution technique is based on an explicit projection method (Peyret and Taylor, 1983) close to the one initially developed for the SURFER code (Lafaurie et al., 1994). Gerris was developed by merging a quad/octree implementation of AMR method with a Volume of Fluid (VOF) approach for solid boundaries and the MAC projection method (Popinet, 2003). The 3D Gerris model was used to study air turbulence associated with a complex shape with good match to observations (Popinet et al., 2004). The *Ocean* module of Gerris was described by Popinet and Rickard (2004) as an adaptive, finite-volume, 3D, incompressible, N-S fluid solver extended into a dynamical core able to model geophysical fluid flows. They demonstrate the accuracy of the 2D model for geostrophic adjustment, a wind-driven circular ocean, and a coastally trapped

Manuscript approved October 16, 2012.

wave. The 3D model was tested for stratified flow over a Gaussian bump. They also show an example of adaptive barotropic flow in a complex coastline and bathymetry, Cook Strait in New Zealand. Rickard and Popinet (2007) demonstrate the application of the 2D solver in Gerris to internal wave breaking. They use a rigid lid because the CFD solver does not allow a free surface. O'Callaghan et al. (2010) followed this demonstration with an application of the 2D CFD solver to the transient behavior of a buoyant plume at both laboratory and field scales. This work directly applies to the use of Gerris to simulate the dynamics of the turbidity maximum as observed in estuaries and the continental shelf. Gerris has also proven useful for wave modeling (Popinet et al., 2010), as well as tsunamis (Popinet 2011; 2012).

The OMEGA model (Bacon et al., 2000; Boybeyi et al., 2001) took a different approach to adaptivity with a 3D mesh that is unstructured in the horizontal horizontal using triangular prisms. It solves 3D nonhydrostatic equations using a flux-based finite-volume method. Other examples of adaptive modeling are described by Jablonowski et al. (2006) and Penner et al. (2005; 2007). The best-known example of a finite-element adaptive mesh ocean model is the Imperial College Ocean Model (ICOM), which has been demonstrated for the lock-exchange test by Heister et al. (2011).

Objectives

This report describes the effort to implement Gerris at NRL for solving problems in estuaries. The relevant processes include the interaction of clay particles with turbulent flows driven by tides and waves. This study thus reinforces the lesson that understanding sedimentation processes necessitates first recognizing the importance of hydrodynamics. This effort has thus turned to AMR and FVM to solve this coupled multi-scale, multi-physics problem in a robust and consistent manner. The work consists of two tasks: (1) implementing Gerris; and (2) application of the model system to problems of interest.

Cited References

D. P. Bacon, N. N. Ahmad, et al. (2000), A dynamically adapting weather and dispersion model: The Operational Multiscale Environment Model with Grid Adaptivity (OMEGA). *Monthly Weather Review*, 128 (7), 2044-2076.

M. J. Berger and A. Jameson (1985), Automatic adaptive grid refinement for the Euler equations. *Amer. Inst. Aeronautics and Astronautics Journal*, 23 (4), 561-568.

S. J. Billett and E. F. Toro (1996), Implementing a three-dimensional finite volume WAF-type scheme for the Euler equations. In J. A. Desideri, C. Hirsch, P. LeTallec, M. Pandolfi, and J. Periaux (eds.) Computational Fluid Dynamics 96*, 732-738.*

A.F. Blumberg and G.L. Mellor (1987), A description of a three-dimensional coastal ocean circulation model. In N.S. Heaps (Ed.), *Three-Dimensional Coastal Ocean Models, Vol. 4*, American Geophysical Union, Washington, DC, 1–16.

Z. Boybeyi, N. N. Ahmad, D. P. Bacon, T. J. Dunn, M. S. Hall, P. C. S. Lee, R. A. Sarma, and T. R. Wait (2001), Evaluation of the Operational Multiscale Environment Model with Grid Adaptivity against the European Tracer Experiment. *J. Applied Meteorology*, 40 (9), 1541-1558.

R. Ford, C. C. Pain, M. D. Piggott, et al. (2004), A nonhydrostatic finite-element model for three-dimensional stratified oceanic flows. Part I: model formulation. *Monthly Weather Review*, 132, 2816 - 2831, doi:10.1175/MWR2824.1.

H.R. Hiester, M.D. Piggott, and P.A. Allison (2011), The impact of mesh adaptivity on the gravity current front speed in a two-dimensional lock-exchange. *Ocean Modelling*, 38 (1–2), 1-21, doi:10.1016/j.ocemod.2011.01.003.

C. Jablonowski, M. Herzog, et al. (2006), Block-structured adaptive grids on the sphere: Advection experiments. *Monthly Weather Review*, 134 (12), 3691-3713.

C. P. Jelesnianski (1966), Numerical computation of storm surges without bottom stress. *Monthly Weather Review*, 94 (6), 379-394.

B. Lafaurie, C. Nardone, R. Scardovelli, S. Zalesdi, and G. Zanetti (1994), Modelling merging and fragmentation in multiphase flows with SURFER. *J. Comput. Phys.*, 113, 134-147.

J. J. Leendertse, R. C. Alexander, and S.-K. Liu (1973), A three-dimensional model for estuaries and coastal seas: Volume I, Principles of computation. RAND Report R-1417-OWRR, 57 pp.

J. O'Callaghan, G. Rickard, S. Popinet, and C. Stevens (2010), Response of buoyant plumes to transient discharges investigated using an adaptive solver. *J. Geophys. Res.*, 115, C11025, doi:10.1029/2009JC005645.

J. E. Penner, N. Andronova, et al. (2007), Three dimensional adaptive mesh refinement on a spherical shell for atmospheric models with Lagrangian coordinates. *SciDAC 2007: Scientific Discovery through Advanced Computing*, 78, U546-U550.

J. E. Penner, M. Herzong, et al. (2005), Development of an atmospheric climate model with self-adapting grid and physics. *SciDAC 2007: Scientific Discovery through Advanced Computing*, 16, 353-357.

R. Peyret and T. D. Taylor (1983), *Computational Methods for Fluid Flow*, Springer, New York, 1983.

S. Popinet (2011), Quadtree-adaptive tsunami modelling. *Ocean Dynamics*, 61 (9), 1261-1285, doi: 10.1007/s10236-011-0438-z.

S. Popinet (2012), Adaptive modelling of long-distance wave propagation and fine-scale flooding during the Tohoku tsunami. *Natural Hazards and Earth System Sciences*, 12 (4), 1213-1227, doi: 10.5194/nhess-12-1213-2012.

S. Popinet, R. M. Gorman, G. J. Rickard, and H. L. Torman (2010), A quadtree-adaptive spectral wave model. *Ocean Modelling*, 34, 36-49.

S. Popinet and G. Rickard (2007), A tree-based solver for adaptive ocean modelling. *Ocean Modelling*, 16, 224-249.

S. Popinet, M. Smith, and C. Stevens (2004), Experimental and numerical study of the turbulence characteristics of airflow around a research vessel. *J. Atmos. and Oceanic Tech.*, 21, 1575-1589.

S. Popinet and S. Zaleski (1999), A front-tracking algorithm for accurate representation of surface tension. *Int. J. Numerical Methods in Fluids*, 30, 775-793.

G. Rickard, J. O'Callaghan, and S. Popinet (2009), Numerical simulations of internal solitary waves interacting with uniform slopes using an adaptive model. *Ocean Modelling*, 30, 16-28.

X. Q. Yin, F. L. Qiao, C. S. Xia, X. G. Lu, and Y. Z. Yang (2010), Reconstruction of eddies by assimilating satellite altimeter data into Princeton Ocean Model. *Acta Oceanologica Sinica*, 29 (1), 1-11, doi: 10.1007/s13131-010-0001-7.

O. C. Zienkiewicz, M. Watson, and Y. K. Cheung (1966), Stress analysis by finite element method-Thermal effects. *Nuclear Engineering and Design*, 4 (5), 498-504.

Publications and Presentations

T. R. Keen and T. J. Campbell (in prep). Tidal dynamics in the Tamar River, United Kingdom. *J. Geophys. Res.*

T. R. Keen, J. D. Dykes, and T. J. Campbell (in prep). Simulations of tidal circulation in Mississippi Bight with an adaptive mesh solver, Gerris. *J. Coastal Res.*

T. Keen, T. Campbell, and J. Dykes (in prep). Comparison of tidal simulations using linear and nonlinear free surfaces in the Gerris Flow Solver. *Ocean Modelling*.

T. Keen, T. Campbell, and J. Dykes (in prep). Model coupling using the Gerris Flow Solver. *Ocean Modelling*.

T. R. Keen, T. J. Campbell, J.D. Dykes, and P. J. Martin (submitted). Gerris Flow Solver: Implementation and application. NRL Memorandum Report MR 7320-12-9441.

T. Keen, T. Campbell, and J. Dykes (in prep). Tidal simulations in macro-tidal estuaries. NRL Memorandum Report MR 7320-??-????

T. Keen and T. Campbell (2012). Simulating the estuary turbidity maximum with an adaptive mesh CFD model (Gerris): Ocean Sciences, Salt Lake City, UT, Feb 2012

T. Keen, T. Campbell, and J. Dykes (2012). Geospatial Analysis of Tidally Driven Flow in Mississippi Sound, U.S.A.: EGU, Vienna, April 2012.

Timothy Keen, James Dykes, and Timothy Campbell (2012). Multiphysics and multiscale model coupling using Gerris: AGU Fall Meeting, San Francisco, Dec 2012.

Document Organization

- Section 1: Project overview
- Section 2: Method of acquiring Gerris and the GTS libraries
- Section 3: GFS Plug-ins
- Section 4: Creating a simulation domain
- Section 5: Boundary conditions
- Section 6: Input/Output processing and analysis with GIS
- Section 7: Testing and evaluation

- Section 8: Setting up the model simulations
- Section 9: Lock-exchange simulations
- Section 10: Example Applications
- Section 11: Appendices

Section 2: Implementation

The Gerris environment consists of three main parts: the Gerris solver itself, a visualization application GfsView, and the Gnu Triangulated Surface (GTS) Library. The Gerris solver does not need interactive display and can run purely in terminal mode. This is useful when running applications on supercomputing systems which are often used in batch mode.

The Gerris solver depends on the GTS library for geometrical operations and object-oriented programming. The GTS library in turns depends on the Glib library, a set of useful extensions for C programming. Glib is installed as part of the standard installation on many Linux systems, however the corresponding development files (library header files etc...) usually need to be installed explicitly.

Additional information about obtaining source code and installation can be acquired from http://gfs.sourceforge.net/wiki/index.php/Installation_summary. A syntax reference is available at http://gerris.dalembert.upmc.fr/gerris/reference/index.html.

Local Build Information

Local versions of the Gerris stable source are maintained in /u/gfs/src. The darcs version control system is used to obtain the source code and updates directly from the Gerris stable repository. In the top-level of /u/gfs/src are the following scripts used to maintain the Gerris source and builds. Usage information for each script is obtained by invoking with the "-h" option.

```
run_checkout : Check-out the Gerris packages from the darcs repositories.
run_update : Download updates (patches) from the darcs repositories.
run_install : Build and install the Gerris packages.
run_clean : Clean out build files from the source directories.
```

Local builds of Gerris stable are maintained in /common/gfs. Date tags are added to keep a history of available builds. Symbolic links are used to point to the latest build. For example,

```
/common/gfs/20120605
/common/gfs/bin -> 20120605/bin
/common/gfs/include -> 20120605/include
/common/gfs/lib -> 20120605/lib
/common/gfs/logs -> 20120605/logs
/common/gfs/share -> 20120605/share
```

Creating a User-Specific Build

The scripts in **/u/gfs/src** can be used to setup a user specific build of GFS. This can be done by either invoking the scripts directly or by making a local copy. Keep in mind that the usage of any of the scripts is obtained by invoking with the "-h" option. This section describes a method that includes making a local copy. The scripts are maintained as a darcs repository. Hence, one can use darcs to get a versioned copy of the scripts and a demo. This can be done with the following.

```
%> darcs get --set-scripts-executable /u/gfs
```

This will create a gfs directory in the working directory that is a local copy of the /u/gfs repository. This includes the src and demo subdirectories. The scripts can now be used to get a copy of the stable or developmental GFS packages. This example will focus on getting the stable version of GFS. To get the stable packages of GFS do the following.

```
%> cd src
%> run_checkout -l stable
```

The "-l" option is for a "lazy checkout", i.e., patch files will only be downloaded as needed. For a full checkout of all patch files do not include the "-l" option. Once the checkout is completed there will e three source directories in src: gerris, gfsview, and gts. To build and install GFS do the following.

```
%> run_install -x /common/hypre
```

This will install GFS into ${HOME}/gfs/YYYYMMDD, where YYYYMMDD is the current date. Symbolic links will be created (for bin, lib, etc...) that point to the subdirectories in YYYYMMDD. The "-x /common/hypre" option sets the build environment to point to the /common/hypre installation so that GFS will be compiled with HYPRE. HYPRE provides an optional higher performance multigrid solver. Options are available for more refined control of the build and installation. The run_clean script can be used for cleaning out the build files in the source directories. The run_update script can be used to query and pull patches from the GFS repositories.

An alternate approach is to use the **/u/gfs/src** scripts directly. This involves first creating the gfs directory and src subdirectory. After which the checkout and install steps (described above) can be followed. For example,

```
%> mkdir -p gfs/src
%> cd gfs/src
%> /u/gfs/src/run_checkout -l stable
%> /u/gfs/src/run_install -x /common/hypre
```

Settings for Run Environment

In this section the bash environment is assumed. If one is using csh/tcsh, then change the syntax accordingly. It is useful to define the following environment variable in the top-level dot file for the environment.

```
export GFS_DIR=/common/gfs
```

To run Gerris executables and access man pages one needs to add the following to the PATH and MANPATH settings.

```
export PATH=$GFS_DIR/bin:$PATH
export MANPATH=$GFS_DIR/man:$MANPATH
```

Gerris relies on pkg-config for obtaining and setting information about the installed libraries. The following environment settings are required for running the Gerris executables.

```
export PKG_CONFIG_PATH=/usr/lib64/pkgconfig:${GFS_DIR}/lib/pkgconfig
```

The GFS Terrain module requires a search path (unless the path is specified in the GFS file) to find terrain databases. The search path is specified using the GFS_TERRAIN_PATH environment variable. For example, one would put the following in a Gerris run script.

```
GFS_TERRAIN_PATH=/u/gfs/topo/global
GFS_TERRAIN_PATH=/u/gfs/topo/regional:$GFS_TERRAIN_PATH
export GFS_TERRAIN_PATH
```

Local Batch System

In the batch system there are 8 core nodes and 12 core nodes available. Depending on the number of cpus you want, you can set ncpus_per_node to either 8 or 12. The best utilization is having nprocs set to a multiple of ncpus_per_node. Batch jobs must be submitted and monitored from stennis. SSH to stennis and then you can use any of following commands.

To check status of jobs

```
%> qstat
```

To peek at the log while running

```
%> qpeek <job number>
```

To watch the log while running

```
%> qpeekf <job number>
User <cntl-c> to quit
```

To remove a job

```
%> qdel <job number>
```

A Demo

In **/u/gfs/demo** is an example gfs input and associated scripts to demonstrate how to run gerris and gfsview.

Package Dependencies for GTS, Gerris, & GfsView

This section describes the required and optional packages for GFS. The package names are based on an Enterprise Linux (EL) or Scientific Linux (SL) distribution. The following packages are required.

```
glib2-devel
netpbm-devel
m4
proj-devel
```

```
gsl-devel
gtk2-devel
gtkglext-devel
startup-notification
mesa-libOSMesa-devel
openmpi-devel
darcs
```

The following packages are optional.

```
netcdf-devel
ode-devel
fftw-devel
hypre-devel
lis-devel
ftgl-devel
ffmpeg
gifsicle
```

On the EL and SL systems use "yum info <package>" to obtain more detailed information on the packages listed above. The darcs package is required for obtaining the GFS code from the source code repositories. Optionally, the source code may be downloaded from **http://gfs.sourceforge.net/wiki/index.php/Download**.

Note that on the NRL systems OpenMPI is not installed via the package manager. Instead, OpenMPI is installed in **/common/openmpi using the ~tjcamp/bin/install_openmpi script.**

Note that the hypre package is not available on EL or SL systems. On the NRL systems hypre is installed in **/common/hypre** using the **~tjcamp/bin/install_hypre** script.

The optional Gerris modules may require additional dependencies (shown in the optional packages list). The dependencies will be checked by the ./configure script and the corresponding modules will only be installed if they are present. A summary of which modules will be installed is given by ./configure. To find out why a particular module is not going to be installed, you need to check further up in the ./configure output which particular library failed to be detected.

Details on Package Dependencies

This section provides details on the packages listed above. This information is obtained using "yum info <package>." Some of the information returned by yum info has been removed. Also, the description section for the "-devel" packages includes description output for the non-devel version of the package. This was done because usually the description provided by the "-devel" version usually only stated that it was a package providing files needed for development.

```
Name        : glib2-devel
Version     : 2.22.5
Release     : 6.el6
Summary     : A library of handy utility functions
```

URL : http://www.gtk.org

Description : GLib is the low-level core library that forms the basis for projects such as GTK+ and GNOME. It provides data structure handling for C, portability wrappers, and interfaces for such runtime functionality as an event loop, threads, dynamic loading, and an object system. The glib2-devel package includes the header files for version 2 of the GLib library.

Name : netpbm-devel

Version : 10.47.05

Release : 11.el6

Summary : Development tools for programs which will use the netpbm libraries

URL : http://netpbm.sourceforge.net/

Description : The netpbm package contains a library of functions which support programs for handling various graphics file formats, including .pbm (portable bitmaps), .pgm (portable graymaps), .pnm (portable anymaps), .ppm (portable pixmaps) and others. The netpbm-devel package contains the header files and static libraries, etc., for developing programs which can handle the various graphics file formats supported by the netpbm libraries.

Name : m4

Version : 1.4.13

Release : 5.el6

Summary : The GNU macro processor

URL : http://www.gnu.org/software/m4/

Description : A GNU implementation of the traditional UNIX macro processor. M4 is useful for writing text files which can be logically parsed, and is used by many programs as part of their build process. M4 has built-in functions for including files, running shell commands, doing arithmetic, etc. The autoconf program needs m4 for generating configure scripts, but not for running configure scripts.

Name : proj-devel

Version : 4.7.0

Release : 1.el6.rf

Summary : Header files, libraries and development documentation for proj.

URL : http://trac.osgeo.org/proj/

Description : Proj and invproj perform respective forward and inverse transformation of cartographic data to or from cartesian data

with a wide range of selectable projection functions. This package contains the header files, static libraries and development documentation for proj. If you like to develop programs using proj, you will need to install proj-devel.

Name : gsl-devel

Version : 1.13

Release : 1.el6

Size : 1.2 M

Summary : Libraries and the header files for GSL development

URL : http://www.gnu.org/software/gsl/

Description : The GNU Scientific Library (GSL) is a collection of routines for numerical analysis, written in C. The gsl-devel package contains the header files necessary for developing programs using the GSL (GNU Scientific Library).

Name : gtk2-devel

Version : 2.18.9

Release : 6.el6

Summary : Development files for GTK+

URL : http://www.gtk.org

Description : GTK+ is a multi-platform toolkit for creating graphical user interfaces. Offering a complete set of widgets, GTK+ is suitable for projects ranging from small one-off tools to complete application suites. This package contains the libraries amd header files that are needed for writing applications with the GTK+ widget toolkit. If you plan to develop applications with GTK+, consider installing the gtk2-devel-docs package.

Name : gtkglext-devel

Version : 1.2.0

Release : 11.el6

Summary : Development tools for GTK-based OpenGL applications

URL : http://gtkglext.sourceforge.net/

Description : GtkGLExt is an OpenGL extension to GTK. It provides the GDK objects which support OpenGL rendering in GTK, and GtkWidget API add-ons to make GTK+ widgets OpenGL-capable. The gtkglext-devel package contains the header files, static libraries, and developer docs for GtkGLExt.

Name : startup-notification

Version : 0.10

Release : 2.1.el6

Summary : Library for tracking application startup

URL : http://www.freedesktop.org/software/startup-notification/

Description : This package contains libstartup-notification which implements a startup notification protocol. Using this protocol a desktop environment can track the launch of an application and provide feedback such as a busy cursor, among other features.

Name : mesa-libOSMesa-devel

Version : 7.11

Release : 3.el6

Summary : Mesa offscreen rendering development package

URL : http://www.mesa3d.org

Description : Mesa offscreen rendering development package

Name : openmpi-devel

Version : 1.5.3

Release : 3.el6

Summary : Development files for openmpi

URL : http://www.open-mpi.org/

Description : Open MPI is an open source, freely available implementation of both the MPI-1 and MPI-2 standards, combining technologies and resources from several other projects (FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI) in order to build the best MPI library available. A completely new MPI-2 compliant implementation, Open MPI offers advantages for system and software vendors, application developers, and computer science researchers. For more information, see http://www.open-mpi.org/ . This package contains development headers and libraries for openmpi

Name : darcs

Version : 2.4.4

Release : 3.el6

Summary : David's advanced revision control system

URL : http://www.darcs.net/

Description : Darcs is a revision control system, along the lines of CVS or arch. That means that it keeps track of various revisions and branches of your project, allows for changes to propagate from one branch to another. Darcs is intended to be an ``advanced *revision control system. Darcs has two* particularly distinctive features which differ from other revision control systems: 1) each copy of the source is a fully functional

branch, and 2) underlying darcs is a consistent and powerful theory of patches.

Name : netcdf-devel

Version : 4.1.1

Release : 3.el6.2

Summary : Development files for netcdf

URL : http://www.unidata.ucar.edu/software/netcdf/

Description : NetCDF (network Common Data Form) is an interface for array-oriented data access and a freely-distributed collection of software libraries for C, Fortran, C++, and perl that provides an implementation of the interface. The NetCDF library also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data. The NetCDF software was developed at the Unidata Program Center in Boulder, Colorado. This package contains the netCDF header files, shared devel libs, and man pages.

Name : ode-devel

Version : 0.11.1

Release : 2.el6

Summary : Development files for ode

URL : http://www.ode.org

Description : ODE is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools. The ode-devel package contains libraries and header files for developing applications that use ode.

Name : fftw-devel

Version : 3.2.1

Release : 3.1.el6

Summary : Headers, libraries and docs for the FFTW library

URL : http://www.fftw.org/

Description : FFTW is a C subroutine library for computing the Discrete Fourier Transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. This package contains header files and development libraries needed to develop programs using the FFTW fast Fourier transform library.

Name : lis-devel

Version : 1.2.53

Release : 3.el6

Summary : Development headers and library for lis

URL : http://www.ssisc.org/lis/index.en.html

Description : Lis, a Library of Iterative Solvers for linear systems, is a scalable parallel library for solving systems of linear equations and standard eigenvalue problems with real sparse matrices using iterative methods. This package contains the development headers and library.

Name : ftgl-devel

Version : 2.1.3

Release : 0.3.rc5.el6

Summary : Development files for ftgl

URL : http://ftgl.wiki.sourceforge.net/

Description : FTGL is a free open source library to enable developers to use arbitraryfonts in their OpenGL (www.opengl.org) applications. Unlike other OpenGL font libraries FTGL uses standard font file formats so doesn't need a preprocessing step to convert the high quality font data into a lesser quality, proprietary format. FTGL uses the Freetype (www.freetype.org) font library to open and 'decode' the fonts. It then takes that output and stores it in a format most efficient for OpenGL rendering. The ftgl-devel package contains libraries and header files for developing applications that use ftgl.

Name : ffmpeg

Version : 0.10

Release : 53.el6

Summary : Hyper fast MPEG1/MPEG4/H263/RV and AC3/MPEG audio encoder

URL : http://ffmpeg.sourceforge.net/

Description : FFmpeg is a very fast video and audio converter. It can also grab from a live audio/video source. The command line interface is designed to be intuitive, in the sense that ffmpeg tries to figure out all the parameters, when possible. You have usually to give only the target bitrate you want. FFmpeg can also convert from any sample rate to any other, and resize video on the fly with a high quality polyphase filter.

Name : gifsicle

Version : 1.60

```
Release     : 1.el6
Summary     : Powerful program for manipulating GIF images and animations
URL         : http://www.lcdf.org/gifsicle/
Description : Gifsicle is a command-line tool for creating, editing, and
getting information about GIF images and animations.
```

Section 3: Gerris Flow Solver Plug-Ins

Introduction

Much of the structure that makes Gerris and GTS so powerful is the implementation of plug-ins. These programs are input by the user in the simulation file and parsed into a *c* programming language function as the file is read. They are then stored on the system and in hash tables to allow their use when Gerris runs. In combination with the linked list and surface representations of the domain, this allows a high degree of flexibility and power to be determined at run time.

GfsFunction

The key mechanism to create *plug-ins* is the implementation in *c* of pointers to non-static member functions as incorporated in *C++*. These pointers need a **hidden argument**, the *this* pointer to an instance of the class. These pointers are implicitly included through the (* *GfsFunctionFunc*) declarations in file, **utils.c**. This method replicates the *this* pointer using *local scope*.

```
typedef gdouble (* GfsFunctionFunc) ( const FttCell     * cell,
                                      const FttCellFace * face,
                                      GfsSimulation     * sim );


typedef gdouble (* GfsFunctionDerivedFunc) (const FttCell     * cell,
                                            const FttCellFace * face,
                                            GfsSimulation     * sim,
                                            gpointer            data );
```

The use of a *typedef* allows GfsFunction*Func to be used as types. The function *templates* implied by these statements require specific arguments: (a) a pointer to an FttCell structure; (b) a pointer to an FttCellFace structure; (c) a pointer to a GfsSimulation structure; and (d) a pointer to data of some kind. Arguments (a) and (b) allow access to the physical domain (grid). Argument (c) includes all of the information associated with a specific simulation. The final argument is used by the derived function type. These functions can be defined by the user at run time to complete a variety of tasks. The GfsFunction class has its initialization function contained in file, **utils.c**.

The construction of a *GfsFunction* object follows the same instantiation paradigm as other *GtsObjects*, but it is fundamentally different from the other classes because it transforms user input to an executable file that is dynamically loaded while Gerris is running. This class constructs a *c*-programming language file from the strings contained in the simulation file, and calls the *system* function to explicitly compile the input into an object file that can be run (i.e., dynamically loaded) from within the calling function. It is placed in the /tmp directory on the file system and is implemented as a *GModule* (i.e., a *plug-in*).

```
struct _GfsFunction {
   GtsObject            parent;
   GString *            expr;
   gboolean             isexpr;
   GModule *            module;
   GfsFunctionFunc      f;
   gchar *              sname;
   GtsSurface *         s;
   GfsCartesianGrid *   g;
   guint                index[4];
   GfsVariable *        v;
   GfsDerivedVariable * dv;
   gdouble              val;
   gboolean             spatial, constant, nomap;
   GtsFile              fpd;
   gdouble              units;
 };
```

There are three steps to generate a plug-in using the GfsFunction class in Gerris:

1) read the expressions that comprise the function from the simulation file

2) build a source code file and compile it as a *c* program

3) create a GModule and store the executable's location in a hash table accessed as a GModule.

All of the functions to complete these steps are contained in the file, **utils.c**. The first step is slightly different for functions that involve constant expressions only, and those that contain variables. There differences will be explored in the examples below.

There is an important distinction to be made at this time with respect to the GfsFunctions associated with file input (e.g., **B_AMP.gts**) and pseudo-code input from the simulation file. The pointer for the *gfs_surface_class_init* assignment of the "read" function (*surface_read*) is the same as the "read" function pointer in *read_simulation*. This pointer is different from the "read" function for the solid class (*gfs_solid_read*). It is not clear what this means but it should be taken as a warning about the unintended use of user-supplied *read* functions.

Read a Function from the Simulation File

The GFS macro processing step replaces any defined substitutions in the simulation file before the file is parsed by *function_read*. An example of an expression with variables is shown on the Appendix A. A constant expression is given below. Note that the expression must include parentheses in order to be parsed correctly. This means that any *Define* statements must include parentheses, even constant expressions.

The resulting expressions between parentheses or brackets "()" or "{}" are appended to the *GString->expr* member in the GfsFunction structure. Thus, the entire function is held in the *module->expression* member of the GfsFunction structure with "\n" used for carriage returns. This expression string will be accessed by the *function_compile* function.

Compose and Compile a GfsFunction

The function, *function_compile*, automatically constructs a *C* source code file using *fprintf* statements. It uses the contents of the GfsFunction structure to determine what constructions are required in the executable. The code it can generate is necessarily limited by these selections. For example, static functions that include ellipses, spheres, and cubes can be included. There is also a header file (**function.h**) that includes a variety of functions used for boundary conditions.

The list of tokens read from the simulation file is compared to the variable list in order to include non-constant functions, including derived variables. The constructed function does not include *while* or *do* loops but there is an *if* block for interpolating variable values to partially wet cells.

The last action by *function_compile* is to call *compile*, which calls the system command to compile the code into an executable file, **/tmp/gfsXXXXXX**, where **XXXXXX** is replaced with a random string generated by the *mkstemp* system function. If this temporary file is not present on the function_cache hash table, it is compiled with a call to compile.

The compiler command is constructed and executed with a *system* call:

```
gcc `pkg-config gerris3D --cflags --libs` -O -Wall -Wno-unused -Werror   \
    -D_GFSLINE_=38 -fPIC -shared -x c /tmp/gfsysGucS -o /tmp/gfsQAjCMX    \
    `sed 's/@/#/g' < /tmp/gfsVOwGzv |  awk '{   if ($1 == "#" && $2 ==    \
    "link") {     for (i = 3; i <= NF; i++) printf ("%s ", $i);   \
    print "" > "/dev/stderr";   }   else if ($1 == "#link")        \
    {     for (i = 2; i <= NF; i++)       \
        printf ("%s ", $i);     print "" > "/dev/stderr";   } else print $0
        > "/dev/stderr";}'        \
        2> /tmp/gfsysGucS ` 2> /tmp/gfsYuoyZp
```

The temporary source file is deleted after successful compilation. The executable file remains on the local file system with its name held in *foutname*.

Create a GModule as a Plug-in

A GModule is created in the *compile* function. The executable file, *foutname* is passed to the GModule function using *gfs_module_new*. This function also places it in the *function_cache* member of the GfsSimulation structure. It is made referential by the *gfs_module_ref* function, which also stores its location (pointer) in the (Gmodule *module) member of the GfsFunction structure. This pointer to the executable is the *key* for the pointers to the GfsModule (GfsModule * m) in the (GfsSimulation * sim)->function_cache hash table.

GfsFunctionConstant Example

The creation of a simple GfsFunction can be demonstrated with a GfsFunctionConstant object, which is a simple application of a GModule. The semidiurnal tidal frequency is defined in the **tides.gfs** simulation file as:

```
Define M2F (2.*M_PI/44700.)
```

This macro is substituted into the simulation file where appropriate. Here are three events for computing harmonic events:

```
EventHarmonic { start = 100000 istep = 10 } P A B Z EP M2F
EventHarmonic { start = 100000 istep = 10 } U AU BU ZU EU M2F
EventHarmonic { start = 100000 istep = 10 } V AV BV ZV EV M2F
```

These lines are read and GfsEvent objects are created. The macro substitution produces a constant expression that is classified as a GfsFunctionConstant object.

In Step (2), these objects are constructed for each line by the function, *function_compile*.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <gfs.h>
double f (void) {
#line _GFSLINE_ "GfsFunction"
  return (2.*M_PI/44700.);
}
```

This is a constant function (i.e., (GfsFunction *f)->parent.klass->info.name = "GfsFunctionConstant").

GfsFunction Example with GfsVariables

A complex tidal statement wit time-dependence is assigned to the *expr* member of the *GfsFunction* structure (i.e., *f->expr*) that was created when *function_read* was entered for a Flather boundary condition.

As with a constant function, *function_compile* is called to create the source code for the function. The includes and declarations are hard-wired in as strings to be placed in the temporary file (e.g., /tmp/gfsRhE3so):

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <gfs.h>
#include <gerris/function.h>
typedef double (* Func) (const FttCell * cell,
                         const FttCellFace * face,
                         GfsSimulation * sim,
                         gpointer data);
double f (FttCell * cell, FttCellFace * face, GfsSimulation * sim) {
  _sim = sim; _cell = cell;
```

The basic variables (GfsVariables) are contained in (GfsDomain * domain)->variables; they are compared one-by-one to (GfsFunction * f)->expr->str, and the result is prepended to a list of variables (GSList *lv). This example contains "B_amp" and "A_amp" as GfsVariables. The same procedure is completed for GfsDerivedVariables and they are added to (GSList *ldv), which contains "t". Their declarations are printed to the temporary file:

```
double B_amp;
double A_amp;
double t;
```

The following lines are hard-coded in *function_compile* if there are any GfsVariables in *lv*:

```
if (cell) {
  B_amp = gfs_dimensional_value (GFS_VARIABLE1 (0x2335600),
         GFS_VALUE (cell, GFS_VARIABLE1 (0x2335600)));
  A_amp = gfs_dimensional_value (GFS_VARIABLE1 (0x2335410),
         GFS_VALUE (cell, GFS_VARIABLE1 (0x2335410)));
} else {
  B_amp = gfs_dimensional_value (GFS_VARIABLE1 (0x2335600),
         gfs_face_interpolated_value (face, GFS_VARIABLE1 (0x2335600)->i));
  A_amp = gfs_dimensional_value (GFS_VARIABLE1 (0x2335410),
         gfs_face_interpolated_value (face, GFS_VARIABLE1 (0x2335410)->i));
}
```

This GfsFunction contains memory addresses (format = %p) instead of pointers as follows: 0x2335600 = address of (GfsVariable *v) containing "B_amp"; 0x2335410 = address of (GfsVariable *v) containing "A_amp". The function *gfs_dimensional_value* returns the dimensional value of the second argument transformed to the dimensions of the first argument. This is where the GfsPhysicalParams function (i.e., L = 185e3) is used to transform the tidal amplitudes because *gfs_dimensional_value* returns "val*pow (L, v->units)". The *gfs_face_interpolated_value* function interpolates to a point that does not coincide with a tidal constituent location. Finally, the GfsDerivedVariable ($v = t$) is assigned a variable:

```
t = (* (Func) 0x7fa1d8152e60) (cell, face, sim, ((GfsDerivedVariable *)
0xa78d50)->data);
```

```
*  0x7fa1d8152e60 = address of the (gpointer func) member of the
   (GfsDerivedVariable * v) object

   (The typedef declaration matches that for the GfsFunctionDerivedFunc from
   utils.h)

* cell = FttCell pointer passed to this function
* face =  FttCellFace pointer passed to this function
* sim = pointer to the GfsSimulation that was passed
* 0xa78d50 = address of the GfsDerivedVariable * v
* data is the (gpointer data) member of a GfsDerivedVariable structure
```

The consequence of this line is to access the model time, *t*. The following lines insert the "GfsFunction" statement into the executable that will be created when function *compile* is called, and return the tidal amplitude in domain units.

```
#line _GFSLINE_ "GfsFunction"
return (A_amp*cos ((2.*M_PI/44700.)*(t))+B_amp*sin ((2.*M_PI/44700.)*(t)));
}
```

The (* GfsFunctionDerivedFunc) type is invoked by functions (*gfs_function_value* and *gfs_function_face_value*) in the following manner:

```
dimensional = (* (GfsFunctionDerivedFunc) f->dv->func) (NULL,
                                                        fa,
                                              gfs_object_simulation (f),
                                                        f->dv->data);
```

where *dimensional* is a gdouble variable, *f->dv* is a GfsderivedVariable pointer member of a GfsFunction structure, and *func* is a gpointer member of the GfsDerivedVariable structure.

GfsModule

Non-static member functions are implemented in Gerris using the GfsModule class, which contains (GfsFunctionFunc f) and (GModule * module) members.

```
typedef struct {
   GModule         * module;
   gchar           * expression;
   guint             refcount;
   GfsFunctionFunc   f;
 } GfsModule;
```

This class accesses the GModule library through its (* module) member. The (gchar * expression) member contains strings used to generate GfsFunctions and the (GfsFunctionFunc f) member serves as the *this* construct discussed above. The GModule *code* is physically constructed using the GfsFunction class.

New GfsModule objects are created using the function, *gfs_module_new*.

```
static GfsModule * gfs_module_new (GtsFile     * fp,
                                   const gchar * mname,
                                   GHashTable  * cache,
                                   const gchar * finname)
{
  GModule * module;
  GfsFunctionFunc f;
  gchar * path = g_module_build_path (GFS_MODULES_DIR, mname);
  module = g_module_open (path, 0);
  g_free (path);
  if (module == NULL)
    module = g_module_open (mname, 0);
  if (module == NULL) {
    gts_file_error (fp, "cannot load module: %s", g_module_error ());
    return NULL;
  }
  if (!g_module_symbol (module, "f", (gpointer) &f)) {
    gts_file_error (fp, "module `%s' does not export function `f'", mname);
    g_module_close (module);
    return NULL;
  }
  GfsModule * m = g_malloc (sizeof (GfsModule));
  m->module = module;
  m->f = f;
  m->refcount = 0;
```

```
  g_assert (g_file_get_contents (finname, &m->expression, NULL, NULL));
  g_hash_table_insert (cache, m->expression, m);
  return m;
}
```

Section 4: Gerris Model Domains

Model domains can be described using simple analytical functions. This section describes more complex methods used in the *GfsOcean* and *GfsRiver* modules.

Defining a Domain with a GTS File

The GTS input is used to define the bathymetry surface for the *GfsOcean* module (3D SWE with linearized free surface). The GTS file format is discussed on the GNU Triangulated Surface file page. The GTS input can also be used to represent data on a surface or curve. This file is accessed continuously by Gerris and it must be efficiently created to facilitate the model running. This will be demonstrated using a working example from Santa Rosa Island, Florida (Figure 4.1). This location is being studied with respect to development of a coupled modeling system in GFS. It was originally simulated using numerous NCOM grids and Shorecirc.

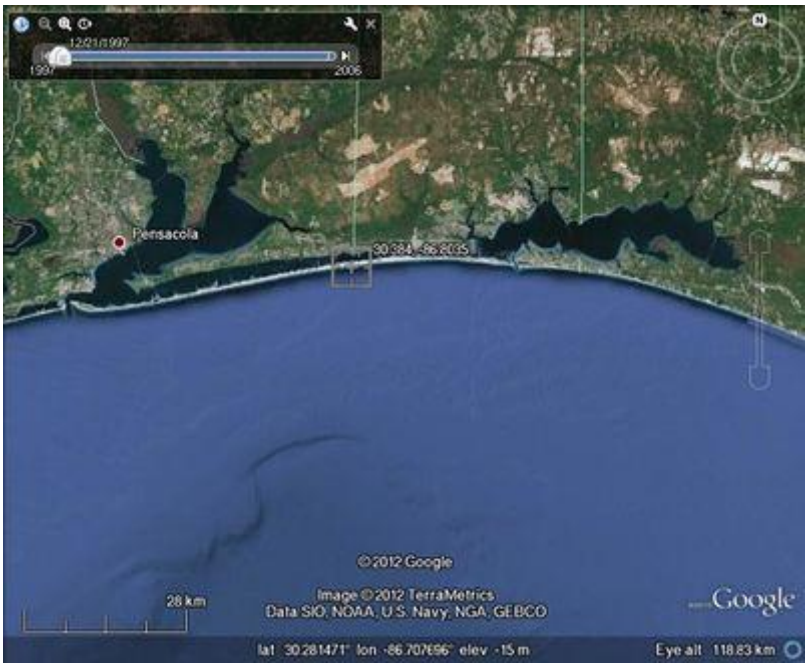


Figure 4.1. Google Earth image of the general area of interest for the Santa Rosa Island simulation with Gerris. The box is located at the exact location of mooring B (Keen and Stavn, 2012).

Creating a GTS Domain File

Several undocumented tests have been completed to arrive at the following guidelines. These impact how the GFS libraries process the bathmetry **gts** file for creating a mesh (e.g., *Solid may not be closed...*), computing the time step (e.g., *segmentation violation in set_timestep...*), computing advection (e.g., *segmentation violation in cell_advection...*), and implementing a boundary condition (e.g., *nothing happens...*). These examples are not exact quotes. The conclusion from these errors is that enough data must be available around the selected domain to ensure that the interpolation process is smooth. There are also potential problems in attempting to over-sample a coarse input bathymetry file.

The *Ocean* module appears to require using the *GfsMap* module. This has only one projection, the Lambert Conformal. The domain is specified by the longitude and latitude of the center of the domain, and its width in meters. The domain can be rotated if desired. Only square domains are computed.

```
PhysicalParams { L = 130e3 }
GModule map
MapProjection { lon = -86.6035 lat = 30.33 angle = 0.0}
Refine 6
```

The refinement specified here is for initialization only. Additional refinement is determined using either *GfsAdapt* or *GfsAdaptFunction* classes with any desired amount of complex refinement implemented as functions. Others are also available.

The **gts** file can be created by using GFS standalone applications to process ASCII files. This begins with the starting files, which in this case consist of a local bathymetry database from SRI and the Gebco 8 minute data. The gebco data are in a standard location. These must be processed into an ASCII file:

```
echo "x1 y1 x2 y2" | kdtquery  /u/gfs/topo/global/gebco/gebco_08 > outfile
```

The **outfile** contains columns of longitude, latitude, and topography (water depth is negative). This file is very useful because it contains all of the land points, which allows Gerris *GfsBoxes* to be located near the coast. The first example uses only the **gebco_08** data. Because of the uncertainty of selecting the appropriate bathymetry, a very large region was selected: $x1 = -90$; $x2 = -80$; $y1 = 25$; and $y2 = 35$ (Simulation 1). This is much larger than the region represented by Figure 4.1 but it definitely works. We have specified $L = 130$ km. We will refer to these ASCII files as **xyz** files.

Before the **xyz** file can be processed further, the water depths must be made positive. The **xyx** file is processed into a triangulated irregular network (TIN) using the *happrox* program.

```
happrox -f -r 1 -c 0.01 < xyz | transform --revert > gts
```

where **xyz** is the file created above and **gts** is the TIN representation of the water depths. The water depths have been made negative again in this file, but modifying these programs (*kdtquery*, *transform*, and *happrox*) is beyond the scope of the current work.

Before a simulation can be run the following environmental variables should be set:

```
PATH    /common/gfs/bin:/common/openmpi/gnu/bin
MPI_DIR /common/openmpi/gnu
GFS_DIR /common/gfs/bin
```

The model is run with the following command:

```
/common/gfs/bin/gerris3D -m waves.gfs |
                 /common/gfs/bin/gfsview3D -s waves.gfv
```

The **waves.gfv** file contains instructions for running *gfsview3D* while *gerris* is running in order to view the results. It must be edited for personal preference.

The simulation was run with different **gts** file descriptions of the domain to achieve the best combination of results and speed. Simulation 1 used the 10° square surface file and was very slow. We need to decrease the size of the **gts** file to improve speed, however. This domain used a 10°×10° bathymetry whereas the required area (Figure 4.2) spans ~1° of longitude and latitude. The **gts** file for Simulation 1 is very slow processing but it allows Gerris to produce the following initial mesh, which uses $R = 6$ to produce a uniform mesh with Δx = 130 km / 64 = 1.5625 km.

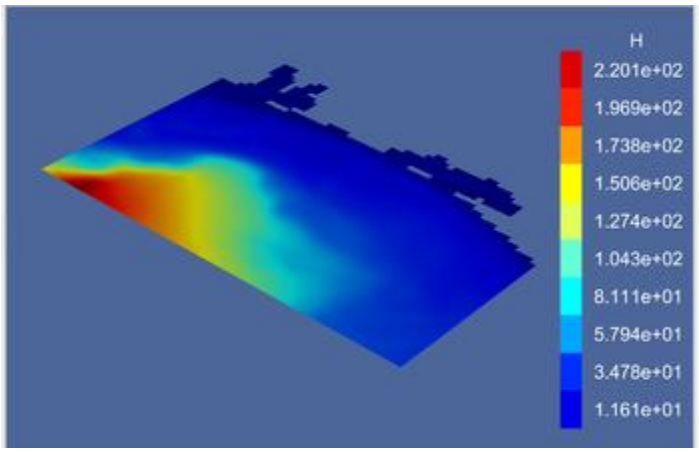


Figure 4.2. Screen dump from GfsView3D showing water depths (Simulation 1).

We next explore the required minimum **gts** file. The size of the **gts** file domain was reduced to "-88.0 28.0 -84.0 32.0" (Simulation 2) for the *kdtquery* command above and the simulation was rerun with no changes to parameters. The resulting bathymetry is the same. When the input to *kdtquery* and subsequently the size of the **gts** file was reduced to "-87.0 -29.0 -85.0 31.0", Gerris failed in *setting the solid fraction from a surface*. This is apparently too small; The

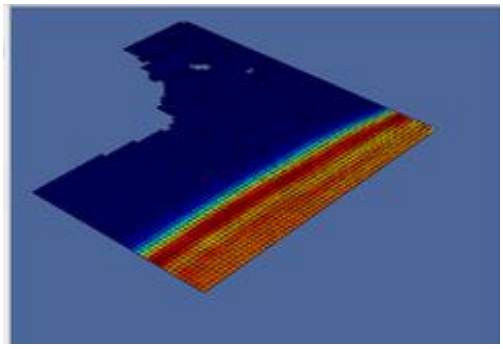
model domain is approximately from (-86.6035, 30.33) and 130 km across, or from -87.1° to -86.1° of longitude and 29.7° to 30.9° of latitude. This explains the failure. A **gts** surface file described by "-88.0 29.0 -85.0 32.0" works; it thus appears reasonable to pad with ~1° (Simulation 4).

Sensitivity Testing for GTS Domains

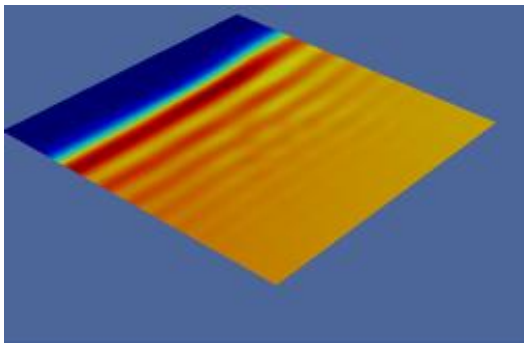
This section discusses implementation issues for the *GfsOcean* module. Two domains are used, as represented by *TIN* files: **beach.gts** and **bight.gts**. The simulations will be done in pairs for the beach, which is a synthetic bathymetry that is processed into a TIN by *happrox*, and the bight, which is the original bathymetry. The waves approach from the east (rt side); the west, north, and south are *GfsBoundary* edges. A higher resolution synthetic bathymetry is in the file, **beach2.gts**.

The large bathymetry file for the Mississippi Bight (MSB) is used as the basis of some small domains in this example (Figure 4.3). This is used to evaluate the synthetic bathymetry used

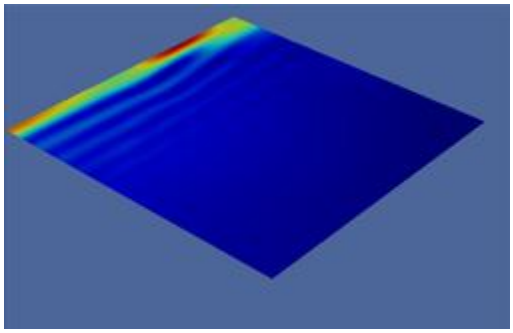
Figure 4.3. Plots of water surface anomaly for short waves (T = 5 s) on the original grid and refinement of 6.



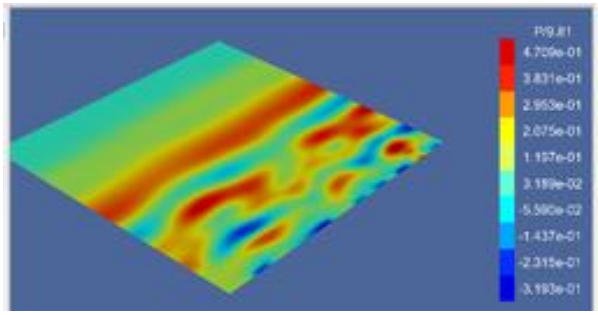
A. After 15 minutes for a domain of 100 km.



B. After 10 minutes for a domain of 10 km.



C. After 15 minutes for a domain of 10 km.

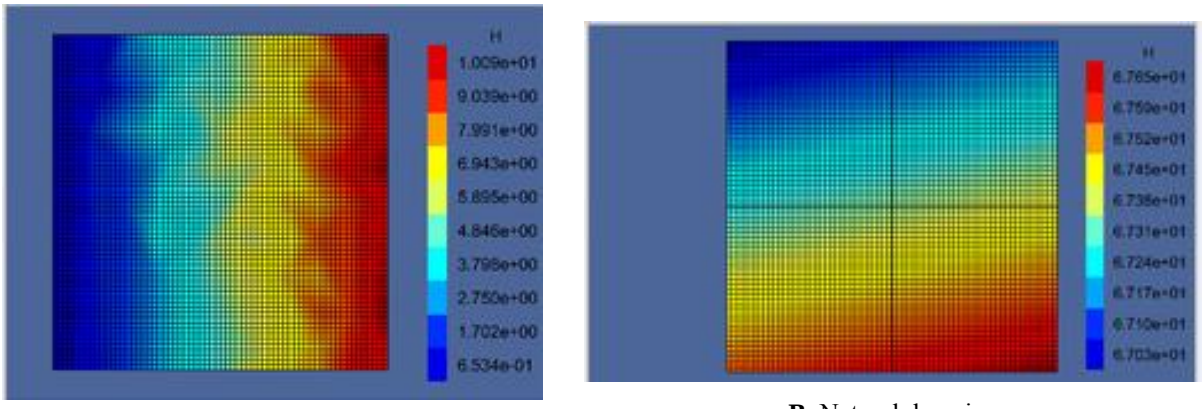


D. After 2 minutes for a domain of 1 km.

The second series of simulations are the same but using the synthetic domain seen Figure 4.4A (center at lon = 271.01° lat = 29.1°). This grid has a cell size of 0.0001° (~100 m) and 2000×2000 cells. The grid based on the real MSB bathymetry (Figure 4.4B) has almost flat

water depths. This has the same mesh size; the only difference should be the files, which are read before assignment of the water depths for each mesh adjustment as well as applying the boundary condition. This has a profound impact on the resulting mesh, boundary condition, and computations.

Figure 4.4. Plots of water depth and mesh for MSB and synthetic domains with refinement of 6 for R = 2^6, Δx = 15 m on 1 km domains.

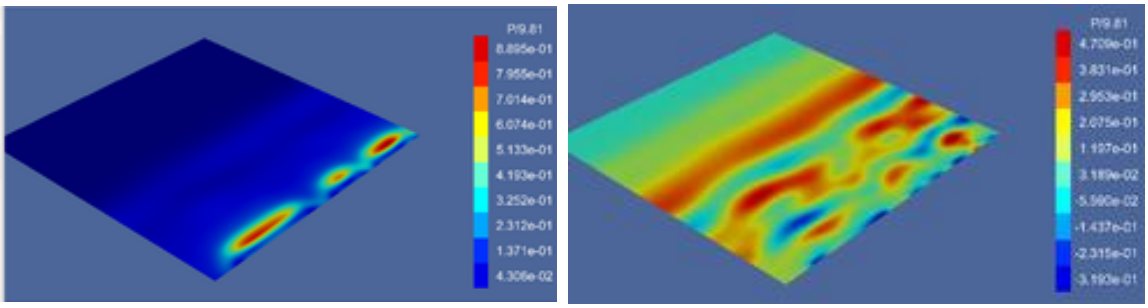


A. Synthetic domain.

B. Natural domain.

The gradient of the synthetic beach is ~10 cm in 100 m or 10^{-3}, which is relatively steep. The max depth is 50 m. The resulting wave field is partly resolved by this mesh size, as seen in Figure 4.5. The linear free surface cannot adequately calculate the wave front and the result is disorganized, even at the boundary (Figure 4.5A). After only 120 s (Figure 4.5B), the wave crests are breaking up. The waves reflect from the steep beach and produced an irregular patter after 15 minutes. However, the maximum water elevation remained below 50 cm. A poorly resolved domain has been found to produce unreal water levels (~3 m) from the incident 14 cm crests. In this case, the errors were associated with an initially low resolution bathymetry that is oversampled for the requested refinement. The irregular waves from Figure 1D are very similar to those from this simulation because of the common problem with the linear free surface.

Figure 4.5. Plots of water level on the synthetic 1 km domain with refinement of 6.

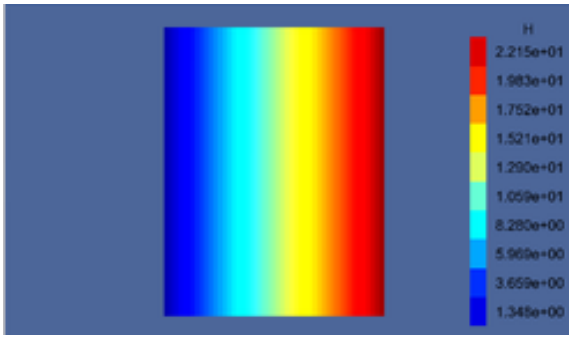


A. 60 s.

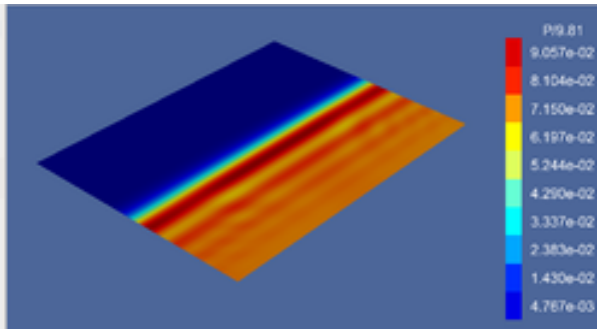
B. 120 s.

A 10 km domain (centered at lon = 271.525°, lat = 29.2°) was simulated using a different initial synthetic bathymetry that had a cell size of 0.001°, or ~ 1 km. The resulting wave field is phase averaged because the same refinement was used (R = 6, Δx = 156 m). The shallowest depths in the grid are 20 cm (Figure 4.6A). The resulting wave field after 5 min. (Figure 4.6B) shows the wave front has a maximum height of 9.5 cm and is propagating smoothly. After 15 min (Figure 4.6C) the wave height has reached 17 cm at the coast from an initial value of 14 cm.

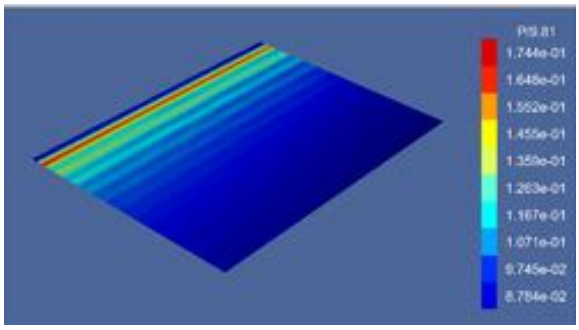
Figure 4.6. Plots of synthetic 10 km domain with refinement of 6.



A. Water depth in meters.



B. Water level (~wave height) at 300 s.



C. Water level (~wave height) at 900 s.

A final test was completed using a very simple input bathymetry. The domain was described by the longitude, latitude, and depth at the four corners of the domain. The result was good for a 10 km domain centered 0.2° from the shoreline (western edge) but the minimum depth was 6 m. When the projection was shifted west to have a minimum depth of 2.5 m, the boundary condition was not applied although the simulation completed smoothy.

Terrain Databases (KDT)

The Gerris Terrain module contains a set of objects which can be used to define solid boundaries using large Digital Terrain Model (DTM) databases. The databases are only limited in size by the amount of disk space available and include a Kd-tree spatial index for efficient retrieval of subsets of the original data.

A Gerris terrain database consists of three files: *basename*.kdt, *basename*.pts, *basename*.sum. Where *basename* is the base name of the terrain database.

The Gerris Terrain databases are usually created with the *xyz2kdt* utility that is available with the Gerris installation. Type "xyz2kdt -h" to see the usage. The basic process involves piping xyz output from a DTM to the xyz2kdt utility.

Here is an example of the steps for creating the ETOPO Gerris terrain database.

```
(1) Unzip the etopo1 package: unzip etopo1_ice_g_i2.zip
(2) Edit etopo_i2_to_xyz.c to make sure the defines match the etopo1 header
    file (etopo1_ice_g_i2.hdr).
(3) Compile: cc etopo_i2_to_xyz.c -o etopo_i2_to_xyz
(4) Run (pipe xyz output to xyz2kdt utility):
    ./etopo_i2_to_xyz < etopo1_ice_g_i2.bin | xyz2kdt -v etopo1_ice_g
```

The kdtquery utility (available with the Gerris installation) can be used to query a KDT database for points that lie within a lon/lat box.

Local KDT Databases

Local KDT databases are maintained in /u/gfs/topo/global and /u/gfs/topo/regional. As the names imply, /u/gfs/topo/global contains global terrain databases and /u/gfs/topo/regional contains regional terrain databases. At the top-level of these directories are symbolic links to the available KDT databases. The symbolic links at the global and regional level simplify the database search path settings for finding KDT databases.

Here is a description of the available global KDT databases.

```
NRL DBDB 2 minute version 4.0:
  Vertical reference: MSL
  Basename: dbdb2.v40
    dbdb2.v40.kdt -> /u/gfs/topo/global/dbdb2/dbdb2.v40.kdt
    dbdb2.v40.pts -> /u/gfs/topo/global/dbdb2/dbdb2.v40.pts
    dbdb2.v40.sum -> /u/gfs/topo/global/dbdb2/dbdb2.v40.sum

ETOPO 1 minute:
  URL: http://www.ngdc.noaa.gov/mgg/global/global.html
  Vertical reference: MSL
  Basename: etopo1_ice_g
    etopo1_ice_g.kdt -> /u/gfs/topo/global/etopo/etopo1_ice_g.kdt
    etopo1_ice_g.pts -> /u/gfs/topo/global/etopo/etopo1_ice_g.pts
    etopo1_ice_g.sum -> /u/gfs/topo/global/etopo/etopo1_ice_g.sum

GEBCO 30 second:
  URL: https://www.bodc.ac.uk/data/online_delivery/gebco
  Vertical reference: MSL
  Basename: gebco_08
    gebco_08.kdt -> /u/gfs/topo/global/gebco/gebco_08.kdt
    gebco_08.pts -> /u/gfs/topo/global/gebco/gebco_08.pts
    gebco_08.sum -> /u/gfs/topo/global/gebco/gebco_08.sum
```

```
GEBCO 1 minute:
  URL: https://www.bodc.ac.uk/data/online_delivery/gebco
  Vertical reference: MSL
  Basename: gebco_1min
    gebco_1min.kdt -> /u/gfs/topo/global/gebco/gebco_1min.kdt
    gebco_1min.pts -> /u/gfs/topo/global/gebco/gebco_1min.pts
    gebco_1min.sum -> /u/gfs/topo/global/gebco/gebco_1min.sum
```

Here is a description of the available regional KDT databases.

```
NOAA Geophysical Data Center (NGDC) Coastal Relief Maps (CRM):
URL: http://www.ngdc.noaa.gov/mgg/coastal/crm.html
Vertical reference: MSL

Northeast Atlantic CRM
  Basename: ne_atl_crm_v1
    ne_atl_crm_v1.kdt -> /u/gfs/topo/regional/ngdc_crm/ne_atl_crm_v1.kdt
    ne_atl_crm_v1.pts -> /u/gfs/topo/regional/ngdc_crm/ne_atl_crm_v1.pts
    ne_atl_crm_v1.sum -> /u/gfs/topo/regional/ngdc_crm/ne_atl_crm_v1.sum

Southeast Atlantic CRM
  Basename: se_atl_crm_v1
    se_atl_crm_v1.kdt -> /u/gfs/topo/regional/ngdc_crm/se_atl_crm_v1.kdt
    se_atl_crm_v1.pts -> /u/gfs/topo/regional/ngdc_crm/se_atl_crm_v1.pts
    se_atl_crm_v1.sum -> /u/gfs/topo/regional/ngdc_crm/se_atl_crm_v1.sum

Eastern Gulf of Mexico CRM
  Basename: fl_east_gom_crm_v1
    fl_east_gom_crm_v1.kdt->
       /u/gfs/topo/regional/ngdc_crm/fl_east_gom_crm_v1.kdt
    fl_east_gom_crm_v1.pts->
       /u/gfs/topo/regional/ngdc_crm/fl_east_gom_crm_v1.pts
    fl_east_gom_crm_v1.sum->
       /u/gfs/topo/regional/ngdc_crm/fl_east_gom_crm_v1.sum

Central Gulf of Mexico CRM
  Basename: central_gom_crm_v1
    central_gom_crm_v1.kdt->
       /u/gfs/topo/regional/ngdc_crm/central_gom_crm_v1.kdt
    central_gom_crm_v1.pts->
       /u/gfs/topo/regional/ngdc_crm/central_gom_crm_v1.pts
    central_gom_crm_v1.sum->
       /u/gfs/topo/regional/ngdc_crm/central_gom_crm_v1.sum

Western Gulf of Mexico CRM
  Basename: western_gom_crm_v1
    western_gom_crm_v1.kdt->
       /u/gfs/topo/regional/ngdc_crm/western_gom_crm_v1.kdt
    western_gom_crm_v1.pts->
       /u/gfs/topo/regional/ngdc_crm/western_gom_crm_v1.pts
    western_gom_crm_v1.sum->
       /u/gfs/topo/regional/ngdc_crm/western_gom_crm_v1.sum

Southern California CRM
  Basename: southern_calif_crm_v1
    southern_calif_crm_v1.kdt->
       /u/gfs/topo/regional/ngdc_crm/southern_calif_crm_v1.kdt
```

```
      southern_calif_crm_v1.pts->
         /u/gfs/topo/regional/ngdc_crm/southern_calif_crm_v1.pts
      southern_calif_crm_v1.sum->
         /u/gfs/topo/regional/ngdc_crm/southern_calif_crm_v1.sum


Central California CRM
   Basename: central_calif_crm_v1
     central_pacific_crm_v1.kdt->
        /u/gfs/topo/regional/ngdc_crm/central_pacific_crm_v1.kdt
     central_pacific_crm_v1.pts->
        /u/gfs/topo/regional/ngdc_crm/central_pacific_crm_v1.pts
     central_pacific_crm_v1.sum->
        /u/gfs/topo/regional/ngdc_crm/central_pacific_crm_v1.sum

Northwest Pacific CRM
   Basename: nw_pacific_crm_v1
     nw_pacific_crm_v1.kdt->
        /u/gfs/topo/regional/ngdc_crm/nw_pacific_crm_v1.kdt
     nw_pacific_crm_v1.pts->
        /u/gfs/topo/regional/ngdc_crm/nw_pacific_crm_v1.pts
     nw_pacific_crm_v1.sum->
        /u/gfs/topo/regional/ngdc_crm/nw_pacific_crm_v1.sum

Hawaii CRM
   Basename: hawaii_crm_v1
     hawaii_crm_v1.kdt -> /u/gfs/topo/regional/ngdc_crm/hawaii_crm_v1.kdt
     hawaii_crm_v1.pts -> /u/gfs/topo/regional/ngdc_crm/hawaii_crm_v1.pts
     hawaii_crm_v1.sum -> /u/gfs/topo/regional/ngdc_crm/hawaii_crm_v1.sum

USGS Gulf of Maine 3 second:
   Vertical reference: MSL
   Basename: gom03_v31
     gom03_v31.kdt -> /u/gfs/topo/regional/gulf_of_maine/gom03_v31.kdt
     gom03_v31.pts -> /u/gfs/topo/regional/gulf_of_maine/gom03_v31.pts
     gom03_v31.sum -> /u/gfs/topo/regional/gulf_of_maine/gom03_v31.sum

Northern Gulf Littoral Initiative (NGLI) 3 second:
   URL: file:///u/gfs/topo/regional/ngli/DATA/ngli_map_bathy_topo.htm
   Domain covered: (-90,29) to (-87,31)
   Vertical reference: MSL
   Basename: ngli_bathy_topo
    ngli_bathy_topo.kdt -> /u/gfs/topo/regional/ngli/ngli_bathy_topo.kdt
    ngli_bathy_topo.pts -> /u/gfs/topo/regional/ngli/ngli_bathy_topo.pts
    ngli_bathy_topo.sum -> /u/gfs/topo/regional/ngli/ngli_bathy_topo.sum

Adriatic 7.5 second:
   Vertical reference: MSL
   Basename: adriatic_7.5sec
     adriatic_7.5sec.kdt-> /u/gfs/topo/regional/adriatic/adriatic_7.5sec.kdt
     adriatic_7.5sec.pts-> /u/gfs/topo/regional/adriatic/adriatic_7.5sec.pts
     adriatic_7.5sec.sum-> /u/gfs/topo/regional/adriatic/adriatic_7.5sec.sum
```

The scripts and programs within the subdirectories of /u/gfs/topo/global and /u/gfs/topo/regional can be used as a guide for generating kdt databases. In each of the database subdirectories in /u/gfs/topo/global and /u/gfs/topo/regional is a program for reading the associated DTM and outputting to stdout the xyz points.

Using The Terrain Module

The Terrain module is initialized by adding the following line to the Gerris parameter file.

```
GModule Terrain
```

The Terrain module defines the following objects.

```
GfsRefineTerrain -- Refines the mesh and creates the corresponding terrain
                    model
GfsTerrain -- Creates a solid boundary following a given terrain model
GfsVariableTerrain -- Defines a variable containing the terrain height
```

The syntax description for the Terrain module is found at http://gfs.sourceforge.net/wiki/index.php/Object_heirarchy#Terrain. There are two ways to specify the search path for KDT databases. One method is to set the path parameter in GfsRefineTerrain or GfsVariableTerrain. For example,

```
GfsRefineTerrain 8 H {
  path = /u/gfs/topo/regional:/u/gfs/topo/global
  basename = gebco_08
} TRUE
```

The path parameter defaults to "." (the local directory) when not specified. This default value can be changed by setting the GFS_TERRAIN_PATH environment variable. For example,

```
GFS_TERRAIN_PATH=/u/gfs/topo/global
GFS_TERRAIN_PATH=/u/gfs/topo/regional:$GFS_TERRAIN_PATH
export GFS_TERRAIN_PATH
```

Terrain Data Base Example

There are several bathymetry sources for this area. It is not available as a single file, but these data have been compiled from NOAA, NGLI, and ETOPO sources. The first step in creating a GFS terrain database is to format all desired sources into the KDT file format, which is readable by the GfsTerrain module. The terrain data are archived in /home/keen/ARCHIVE/grids/gulf_of_mexico:

```
Bay_St_Louis/bay_st_louis_3s.xyz  (~92 m)
gulf_of_mexico/mississippi_sound/ngli_bathy_3s.xyz  (3 arc-second or ~92 m)
```

The third is a standard terrain database from NOAA (**/u/gfs/topo/regional**):

```
central_gom_crm_v1  (3 arc-second or ~92 m)
```

Others are available but these are the best coverage. This section will discuss how it was determined that these were the best to use. The **xyz** files are individually transformed into KDT files using the following commands:

```
cat ngli_bathy_3s.xyz | /common/gfs/bin/xyz2kdt -v ngli_bathy_3s
cat bay_st_louis_3s.xyz | /common/gfs/bin/xyz2kdt -v bay_st_louis_3s
```

Examples of the resulting *tree-based* data are stored in the following files:

```
msgom007_010799.kdt
msgom007_010799.pts
msgom007_010799.sum
```

These preliminary topo/bathy databases are examined using *Gerris'* Shallow Water Module (*GfsRiver*) to create **gfs** files for viewing with *Gfsview2D*. This is necessary because we do not have a utility for viewing the **KDT** terrain files. The general contents can be printed to the screen or sent to a file with *kdtquery* as follows:

```
echo "-90 30 -89 31" |  ngli_bathy_3s
```

This command reveals that there are a lot of land points in addition to water values, which are less than zero. The simulation file, **tides.gfs** was adjusted to produce a reasonable that can be used for the *Ocean* Module. The resulting terrain (**ngli_1s-3s.xyz**) is gridded to ~80 m and is more than large enough (Figure 4.7).

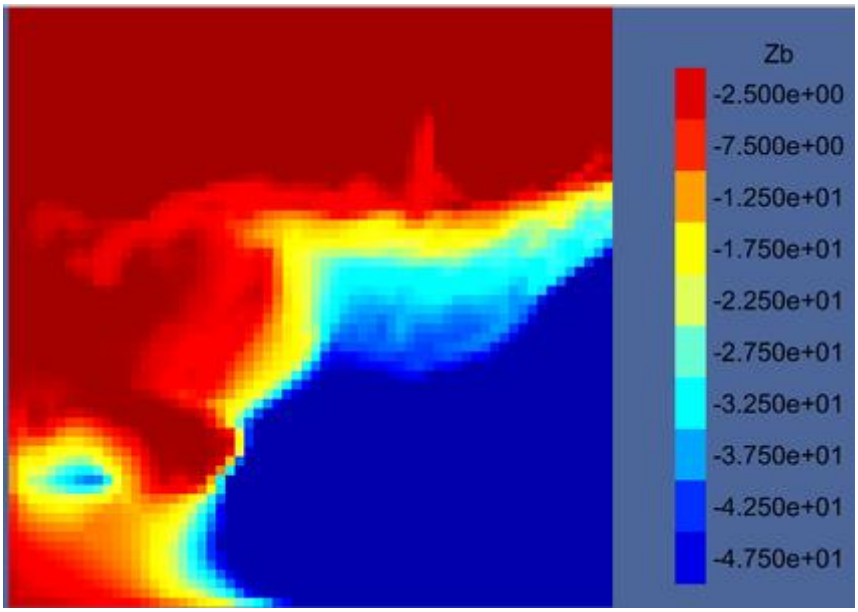


Figure 4.7. Screen shot of gfsview2D showing the NGLI bathy (~80 m) to be used for a GTS file.

But how good is it? We can use the AMR in Gerris to evaluate this terrain in detail. The *GfsAdaptError* class is a subclass of the *GfsAdaptGradient* class, which is a subclass of the *GfsAdapt* (Event) class. The included GfsFunction increases refinement wherever the bottom depth changes rapidly.

```
AdaptError { istart = 0 istep = 1 iend = 1 } {
     cmax = 1.0
```

```
      cfactor = 4
      weight = 1.0
      minlevel = 0
      maxlevel = bathyLEVEL
      maxcells = 10000000
} (Zb <= 0 && Zb > -1500 ? Zb : 0)
```

where cmax = max allowable cell cost; cfactor = divisor for coarsening a cell (i.e., cost is smaller than cmax/cfactor); weight = weight for each factor. The resulting cell plot (Figure 4.8) shows that there is a major discontinuity at ~89° (off the delta) due to using multiple bathymetry with major differences in depths.

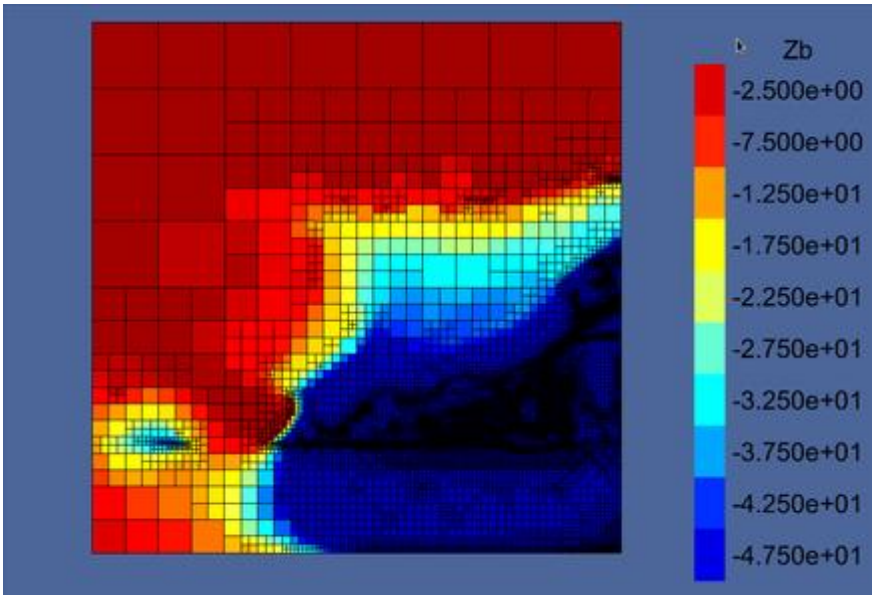


Figure 4.8. Screen shot of gfsview2D showing the NGLI bathy with initial cell refinement for depths <1500 m.

A lower resolution terrain file for the northern gulf of mexico (**ngom05_060612.xyz**) shows a much better likelihood of matching with the higher resolution terrain that was specifically created from the NGLI data (Figure 4.9).

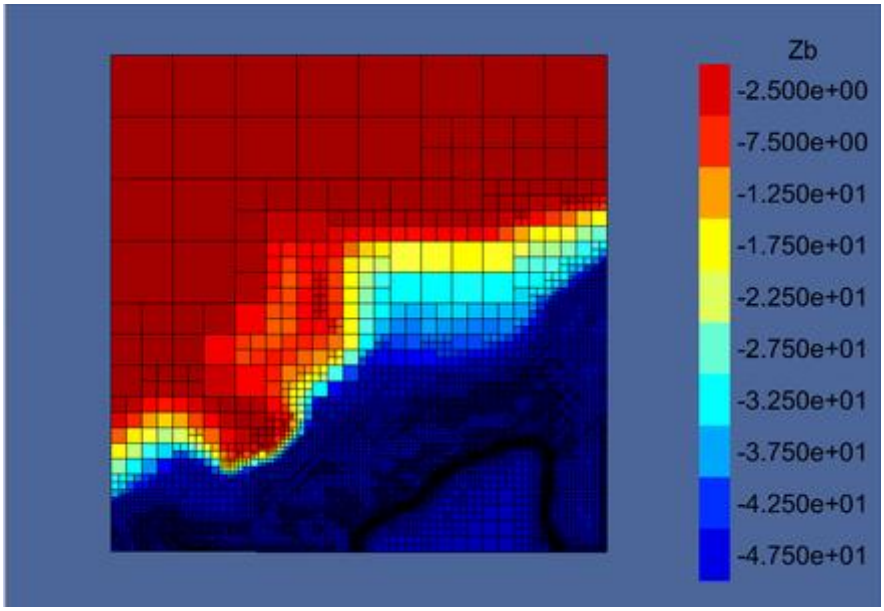


Figure 4.9. Screen shot of gfsview2D showing a low-resolution terrain with initial cell refinement based on depth changes <1500 m.

This terrain is much too coarse for the Mississippi Sound and Bay St. Louis areas, however. We can use a 3 arc-second terrain from NOAA for Bay St. Louis and merge it with the coarser one to get this result (Figure 4.10).

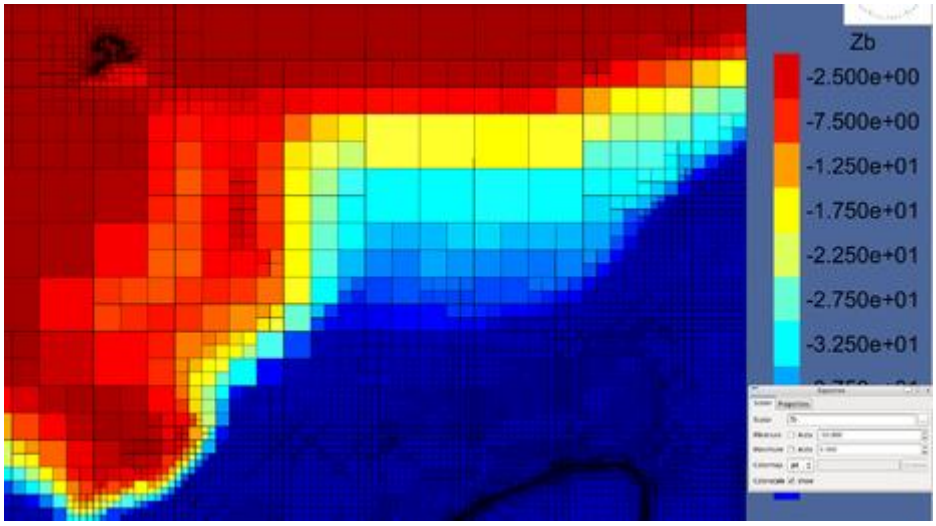


Figure 4.10. Screen shot of gfsview2D showing a low-resolution terrain mixed with a 3 second terrain for Bay St. Louis, with 1-hour cell refinement based on depth curvature and wetting/drying.

These terrains do not overlap and the Miss. Sound area is resolved at the lower resolution. This can be addressed with an intermediate terrain (**msgom_01_040497**) (Figure 4.11).

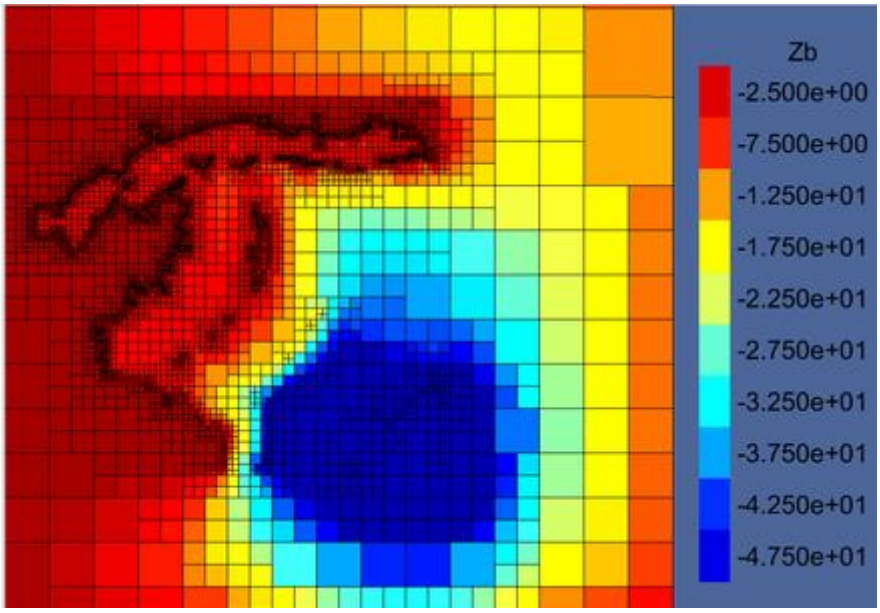


Figure 4.11. Screen shot of gfsview2D showing an intermediate-resolution terrain for the Miss. Bight region, with 1-hour cell refinement based on depth curvature and wetting/drying.

The use of all three terrains demonstrates several aspects of such a merging (Figure 4.12). First, the smallest area of special interest (Bay St. Louis) has been resolved at the highest resolution as desired. However, the mismatch in grids at the SE corner of the intermediate grid is apparent. This is caused by a practice of making a constant depth for small areas with much deeper water, which would impose a small CFL constraint on the model time step otherwise. This practice has apparently worked its way into many of the available bathymetry files.

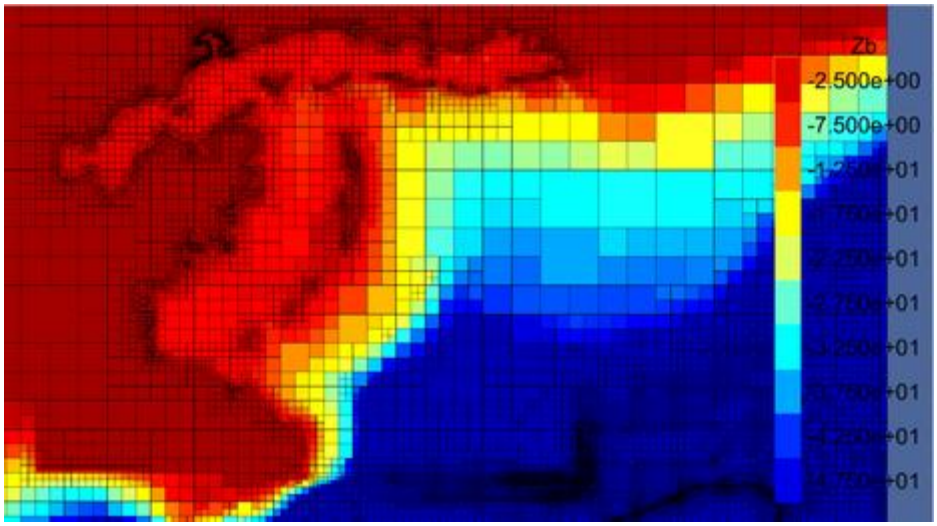


Figure 4.12. Screen shot of gfsview2D showing the result of all three terrains for the Miss. Bight region, with 1-hour cell refinement based on depth curvature and wetting/drying.

One way to address this problem is to remove all depths in the intermediate terrain that are 80 m. This results in only the valid points being available for the merged mesh. Before examining the result, it is useful to use available ground truth, which in this case is available from GoogleEarth because they include altimetry-derived bathymetry in this area. The image below (Figure 4.13) shows salt domes and a sharp drop-off from 60 m to 200 m.

Figure 4.13. Screenshot from Google Earth of Mississippi Bight bathymetry derived from altimetry.

The mesh that results from the modified MS grid (intermediate) lacks the square mismatch area but does indicate a slightly angular gradient in the vicinity of the drop-off east of the delta (Figure 4.14). This is consistent with the altimetry and may be acceptable if it is representative of the terrain rather than the gridding procedure.

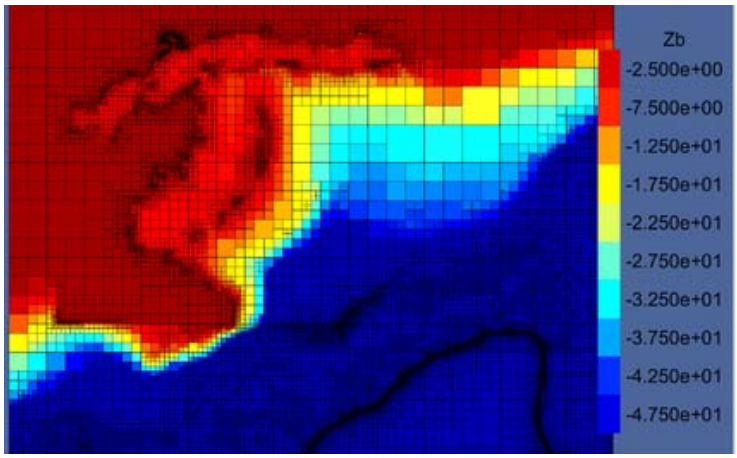


Figure 4.14. Screen shot of gfsview2D showing the result of all three terrains for the Miss. Bight region, with 1-hour cell refinement based on depth curvature and wetting/drying.

There is a problem with this grid, however; it reflects the southern edge of the intermediate terrain and is unrealistic. These problems indicate the difficulties of working with previously gridded bathymetry: (1) it often neglects land and thus cannot be used for intertidal computations; (2) it may include artificial depths that are intended for specific simulations; (3) grids may not overlap and may have serious mismatched terrain values in overlapping areas.

Section 5: Boundary Conditions

Introduction

Gerris implements a number of standard boundary conditions as standard functions. The boundary conditions are introduced through the GfsBc class (Figure 5.1) and they are processed as *GfsInit* events. Note that this is not part of the GTS library. It is used in the ocean module.

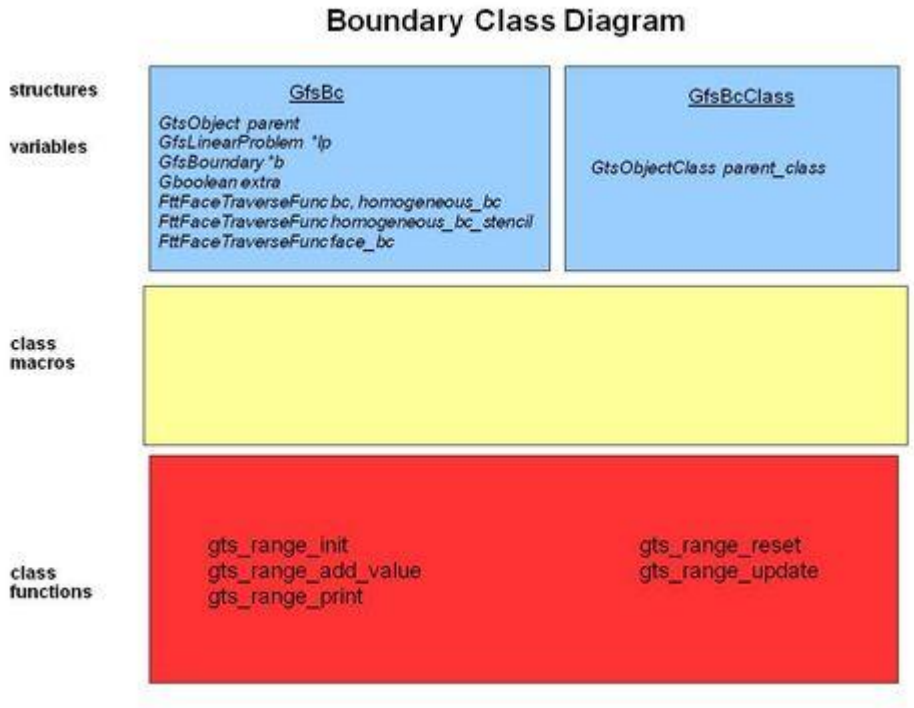


Figure 5.1. Simplified diagram for the GfsBc Class.

The structure *GfsBc* is defined in the **boundary.h** file of GFS and not a part of the tide module (see section below). This is per the standard template constructions within GTS and inherited by GFS. The GfsBc structure contains members:

```
GtsObject           parent;
GfsLinearProblem *  lp;
GfsBoundary *       b;
GfsVariable *       v;
gboolean            extra;
FttFaceTraverseFunc bc, homogeneous_bc;
FttFaceTraverseFunc homogeneous_bc_stencil;
FttFaceTraverseFunc face_bc;
```

This class contains GTS members, so it is a low-level class (sort of) that utilizes existing GTS functionality directly. The function *tide* is associated with a general tide (GfsBc.bc). To summarize,

```
tide calls tide_value, which calls amplitude_value,
  which finally calls fes. This appears to be a direct connection to the
   fes2004
  database. The GfsBc class does not differentiate sources or types of
   boundary
  conditions. This is up to the user.
```

The classes that comprise the *boundary conditions* are illustrated in Figure 5.2.

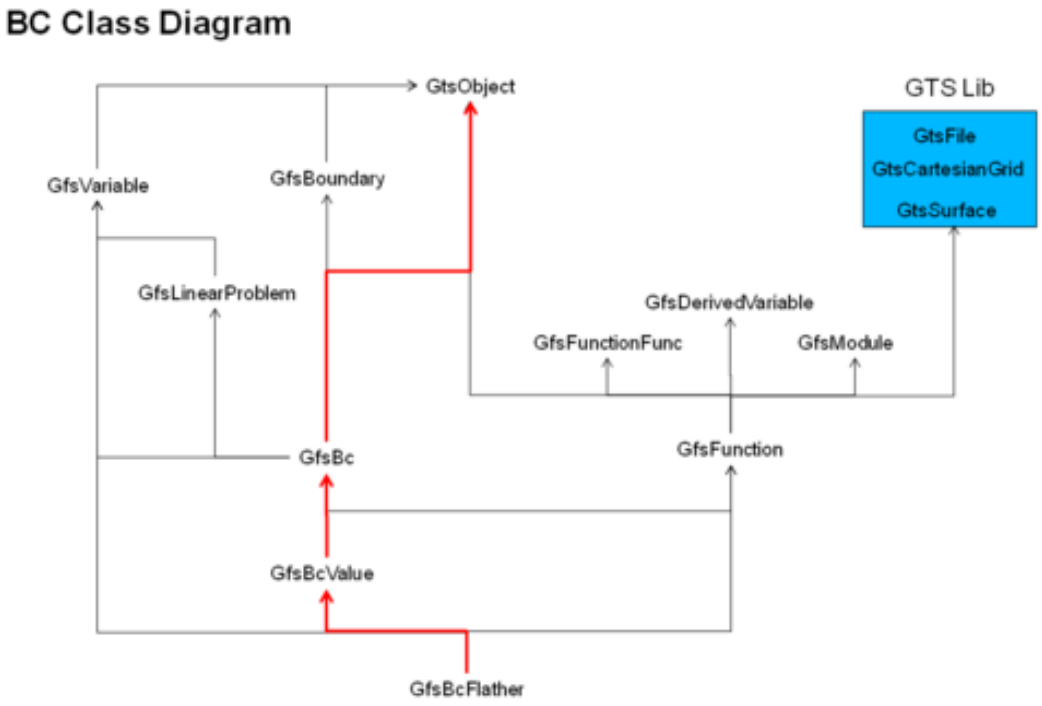


Figure 5.2. Schematic diagram of the relationships between members of the *GfsBcClass*.

Tidal Boundary

Tide processing is started when a GFS class is encountered in the simulation file and its "read" function is invoked. For example, if the string ".gts" is parsed, function *read_surface* (**utils.c**) is called whereas a *float* or *int* is parsed directly. For **gts** file input, the sequence is more complex because the initial processing by *read_surface* only results in the surface data (i.e., *x*, *y*, and *z* values) being placed in containers.

A GModule for processing the tides is not created until the boundary conditions are processed. The constituents are stored as GfsEvent objects until the GfsBcFlather class is processed in function *ocean_run*.

I found a note in ocean.c referring to **modules/tide.mod**, which is used to generate the **tide.c** program. There is no difference between **tide.c** and **tide.mod**. If changes are to be made, however, they need to done in tide.mod. The tide module is not used in the Cook Strait problem because the tide amplitudes are read from files and not imported from the database. This class is not part of the GfsBc class (Figure 5.2).

General Observations in **tide.mod**:

```
Struct _GfsBcTide is defined with members:
  GfsBcValue     parent,
  gdouble **     amplitude, **phase, x, and size;
  GfsVariable *  h, p.
```

P is going to be pressure (or anomaly) and h = water depth, based on the standard used in the simulation input file (**tide.gfs**). The usual procedure is followed with the macros GFS_BC_TIDE and GFS_IS_BC_TIDE.

This section discusses the apparent sequence in which the tidal input is processed in Gerris. Some of this algorithm is also discussed in *simulation file* processing analysis (Appendix A).

There are several steps to processing the tidal data into the model:

```
1. Reading the files
2. Storing the surface information
3. Processing the input GtsSurface into a GfsInit object
4. Processing the GfsInit objects into GfsBcFlather objects
5. Applying the values to the pressure equation along the boundary
```

Constant Values Supplied in Simulation File

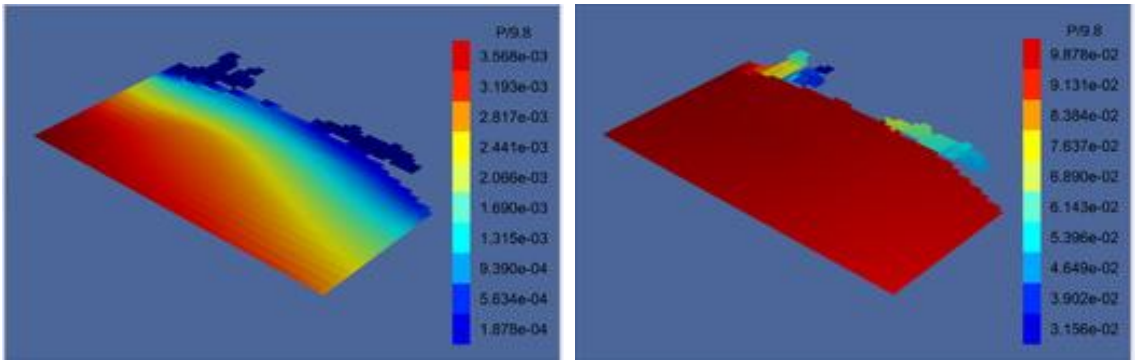
The K_1 tide is approximated using the data from Pensacola (see Keen et al., 2013,MR Report, *Coupled Hydrodynamic and Morphologic Modeling with Gerris*) with amplitude, phase values of 14 cm and 320° *G*. This is implemented as discussed in the Cook Strait example. The key elements as implemented here are:

```
Define RTIME 86400.0
Define RAMP(t) (t > RTIME ? 1.0 : t/RTIME)
Define K1F (2.*M_PI/86162)
Define K1p 320.0
Define K1a 0.14
Define K1(t) (A_amp*cos (K1F*t)+B_amp*sin (K1F*t))
Define TIDE(t) (RAMP(t)*K1(t))
...
Init {} {
    A_amp = K1a*cos(K1p*180./M_PI)
```

```
    B_amp = K1a*sin(K1p*180./M_PI)
    flip = 1
}
...
GfsBox {
    left = Boundary {}
    right = Boundary {}
    bottom = Boundary {
        BcFlather V 0 H P TIDE(t)
    }
    front = Boundary {}
}
```

With the final bathymetry from Simulation 1 (previous section), the tide propagated smoothly with no noise during ramp up (Figure 5.3).

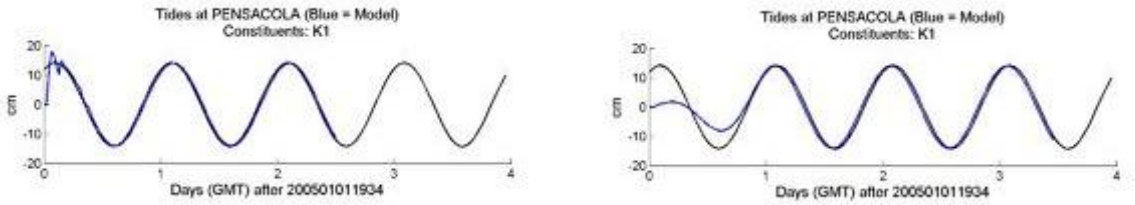
Figure 5.3. Screen dump from GfsView3D showing tide propagation.



A. Hour 1. **B.** Hour 24

The simulation uses a ramp time (*RTIME*) of 1 day so the resulting time series of water levels at Mooring B from Santa Rosa Island have not reached full height. A longer time period (3.5 day) reveals a reasonable match between the model and the K_1 tides at Pensacola (Figure 5.4A). The model has no time, and the match is coincidental. The tide should arrive somewhat earlier on the outer island shoreline than at the tide gauge. The detailed data for Mississippi Sound suggest about 20 minutes (Seim et al., 1987).

Figure 5.4. Time series of K_1 water level at Pensacola tide gauge (black line) and model prediction at the gulf side of Santa Rosa Island (blue line).



A. Simulation 2.	**B.** Simulation 4.

It is not clear why the ramp-up does not appear in Figure 5.4A but it is obvious in Figure 5.4B.

Tides Module (from FES2004 only)

There are two parameters defined as macros: N = 64 (number of discretization points), and NM = 14 (number of tidal modes, which must match FES2004). Fes2004 is a subdirectory of modules. It contains the following files:

- fes2004_alloc.c,
- fes2004_error.c,
- fes2005_extraction.c: this function is passed a filename along with pointers to location, lat, lon, amplitude, and phase. It calls a series of functions directly related to the fes2004 tide database.
- fes2004_init.c,
- fes2004_io.c: opens a netcdf file for each cpu. The name is contained in 'filename'. It passes a pointer to a structure to NC_OPEN as the unit number.
- fes2004_kernel.c,
- fes2004_prediction.c,
- fes.h
- fes2004_lib.h.

The FES 2004 database is a global ocean tide spherical harmonic coefficients.

- F. Lyard, F. Lefevre, T. Letellier, O. Francis, "Modelling the global ocean tides: insights from FES2004," Ocean Dynamics, 56, 394-415, 2006).

This will generate the coefficients file that are referred to in the shell script used for this simulation. File **tide.c** includes the function, *bc_tide_read*. This function is passed a GtsFile and a pointer to an array of GtsObjects. The parameters, N and NM, are used to allocate memory for the amplitude. There is a block to "read embedded coefficients", which loops over N and NM. The file must contain only numbers for amplitude and phase. This is hard-wired for the fes2004 data. I believe this is referring to actual coefficients in the input simulation file instead of the 'A_amp = Am2.gts' statement. The second option is to "extract FES2004 tidal coefficients" using the environmental variable, GFS_FES2004" if it is defined, or the file "tide.fes2004.nc". An error will occur if neither is available.

We need to check where this function (*bc_tide_read*) is called. It is assigned to the "klass->read" member of the GtsObjectClass, which is its parent, in *gfs_bc_tide_class_init* in the usual initialization paradigm. The calling function, *gfs_bc_tide_class*, initializes the gfs_bc_tide_info structure with its name. This function is called in a macro, GFS_BC_TIDE. This macro does not appear to be used to assign the tide amplitude from a file in the simulation file. This macro only appears in tide.c in function *tide_value*, where it is passed as a function argument (standard approach) for the depth to *gfs_face_interpolated_value*; the returned value is assigned

to *H*, which is a double scalar. The amplitude is calculated by another call to GFS_BC_TIDE passed to function *amplitude_value*. This is modified for the current time (sim->time.t + deltat) and corrected for the gravity wave velocity. Note that the reference depth is fixed at 5000 m.

The function *tide_value* is called by *tide*, which is the "bc" member of the structure GfsBc initialized by *gfs_bc_tide_init*. At this point, specialized module names may no longer be used and it may become necessary to track instantiations of the GfsBc structure. There are several occurrences in file tide.c, and "bc" is passed to *amplitude_value*. This function occurred just above. However, if the end result of this line is to call the macro GFS_BC_TIDE, it is intended to extract/read the fes2004 tide solution only.

Extracting tides from a database

The tides will be extracted from the OSU tide database:

```
/home/keen/ARCHIVE/DATA/TYPE_OF_DATA/tides/TIDE_TABLES/osu_tides/"
```

The tides will be processed using a program built from *gts_tides.f*, *tiderot.F*, and *tide_egb.F*. This simulation will only use the tidal heights. The tide extraction is completed for the northern Gulf of Mexico grid clipped to -90<x<-87 and 28<y<31. This reduced the number of points to 3540. The tide extraction program was also modified to use a parameter to select only the tidal heights if desired (ncomp=1). This was necessary because the triangulation procedure was taking far too long for all of the grid points. The output consists of files, **k1_coefficients**, **m2_coefficients**, and **o1_coefficients**. These contain the following lines:

```
      270.0230         28.0479          0.1452         11.8485
      270.0230         28.0979          0.1451         11.7649
      270.0230         28.1479          0.1450         11.681...
```

The columns are longitude, latitude, amplitude, and phase for the given constituent. These files are piped to the *delaunay* program that is part of the GFS software library:

```
print $1 " " $2 " " $3*cos($4*3.14159265357/180.) \
                  < m2_coefficients | delaunay > AM2.gts

print $1 " " $2 " " $3*sin($4*3.14159265357/180.) \
                  < m2_coefficients | delaunay > BM2.gts
```

where $1, $2, $3, and $4 refer to the longitude, latitude, amplitude, and phase columns from the input file, respectively. The format of these files looks like this:

```
3540 10383 6844 GtsSurface GtsFace GtsEdge GtsVertex
272.823 30.6479 -0.0003878
272.773 30.6479 -0.000242959
272.773 30.6979 -0.00012148
272.373 29.7479 -0.00712594
272.323 29.7479 -0.00779957...
```

This should be the correct format to be read by Gerris as a GtsSurface.

Tidal Constituents from GTS Files

I believe that tides read from **gts** files are implemented through the boundary class in GFS and NOT as a module. In directory, src, we find the **boundary.c** file, which has similar functions to the **tide.c** file, except its functions do not include "tide" within their names. This sequence is PROBABLY started by the "GfsBoundary" (sometimes the "Gfs" is not used--optional) in the simulation file.

Boundary heights and/or currents must come from elsewhere. The structure, GfsBcValue has only two members:

```
GfsBc         parent
GfsFunction * val
```

The function "GfsBcValue.val" is initialized in *gfs_bc_value_init* by a call to *gfs_function_new*, which creates (if necessary) a new GfsFunction. However, (GfsFunction *val) has no value yet. The function *gfs_bc_value_class_init* assigns the key structure members:

```
klass->write   = bc_value_write;
klass->read    = bc_value_read;
klass->destroy = bc_value_destroy;
```

We want to look at occurrences of the *bc_value_read* function to look for errors. This function receives an array of pointers to GtsObjects, which represent the boundary condition values, and a pointer to the (PROBABLY) simulation file. The file has been read up to the appropriate line and the keyword "Boundary" will cause this function to be called from some, as yet undetermined, location in the code. There are two read statements in *bc_value_read*. First is the "read" function for the parent class of the GfsBcClass parent, which is a GtsObjectClass. This function is set to NULL by the underlying GtsObjectClass init function. The standard arguments to a GtsObject read function are an array of pointers to structures and a pointer to a file, in this case **o and *fp, which were passed to *bc_value_read*.

The basic read function for a binary gts file is *gts_file_read*, which is a wrapper for the *c* function *fread*. Function *gts_file_read* is located in file, **misc.c**, but the declaration is in **gts.h**. If the gts file is ASCII, function *atof* is called by *gts_point_read* to recast 1 point at a time from the gts file. The points are read by *gts_file_next_token*. There are three calls; the x coordinate, y coordinate, and z coordinate. Function *gts_point_read* is the "read" function for the *point* class, which is a subclass of the *GtsObject* class. The arguments passed to *gts_point_read* are the same as the GtsObjectClass "read" function, **o and *fp. I THINK that the *gts_point_read* function is inherited from the point class by the definition of the "read" function in *gfs_bc_value_class_init*, which invokes the GTS_OBJECT_CLASS macro with the *gfs_bc_value_class* function and its "read" member. In other words, the parent of a boundary value is a point value, which makes perfectly good sense (to me). We need to locate the function that calls *gts_point_read* because it only reads one line.

The GTS tide files are implemented as GfsInit objects included in the simulation file:

```
GfsInit {} {
             A_amp = AM2.gts
             B_amp = BM2.gts
           }
```

Their contents will be placed in a GfsEvent structure that is associated with an object from the pseudoclass, *GfsInit*. The unique operation of Gerris allows these statements to be transformed into a c-function when Gerris runs. This begins with the GfsInit class being read from the file before *gfs_function_read* is called to parse the strings included between the opening "{ " and closing "} " tokens. This is a user-supplied function that is defined in file utils.c. It calls the *read* member of the GfsFunction class' parent (GtsObject), which is *function_read*.

```
gfs_function_read:  (*  GTS_OBJECT  (f)->klass->read)  (&o,  fp);  (read  =
   function_read)
static void function_read (GtsObject ** o, GtsFile * fp)
```

where: &o is a pointer to a GfsFunction and fp is the GfsFile pointer. This function interprets the input from the simulation file as a *c* function and returns a pointer. Within the "Boundary" block we find "BcFlather" defined.

Flather Boundary Condition

If the key word "Boundary" is parsed from the simulation file, a *GfsBoundary* object will be created. Furthermore, if the keyword "Flather" is encountered, a *GfsBcFlather* object will be created. Class *gfs_bc_flather_class*() is initialized as an entry in the array, classes by function gfs_classes. This is its only occurrence in this file (init.c).

The GfsBcFlather class contains the following functions:

```
bc_flather_write
set_gradient_boundary
bc_flather_read
bc_flather_destroy
flather_value
flather
homogeneous_flather
face_flather
gfs_bc_flather_class_init
gfs_bc_flather_init
gfs_bc_flather_class
```

There is a macro defined for the flather bc: GFS_BC_FLATHER, which creates a new GfsFlather object. The GfsBcFlather structure has the following members:

```
GfsBcValue    parent
GfsVariable * h, *p
GfsFunction * val
```

As part of evaluating the functionality of the Mississippi Bight tidal simulations, I have been tracking the processing of the Flather BC.

Methodology

The tidal constituents are implemented through the GfsBox object. Function *gfs_box_read*, reads all of the boundary conditions and a new GfsBc object is created by *gfs_bc_new*. This function is invoked when "Boundary" is parsed from the simulation file.

```
GfsBox {
     left = Boundary {
         BcFlather U 0 H P M2(t)
     }
}
```

The type of boundary condition, "BcFlather", is read by other functions that are called by the *read* member of the GfsBc class, which is *gfs_boundary_read* (assigned in function *gfs_boundary_class_init*). This schematic shows the sequence:

```
gfs_boundary_read>>boundary_read_extra_bc>>gts_file_next_token     /*  reads
  "BcFlather" */
```

This operation identifies the type of boundary condition only. The "BcFlather" BC is processed by function *boundary_read_extra_bc* (in file **boundary.c**). The *read* function for a GfsBcFlather object is *bc_flather_read*. This is initialized by *gfs_bc_flather_class_init*; the functions for this object are contained in file, **ocean.c** because there is no class defined for a GfsBcFlather object. The *boundary_read_extra_bc* function calls the *read* function for the "BcFlather" object in this innocuous statement:

```
(* klass->read) (&object, fp)
```

where: *klass* is the class returned from *gfs_object_class_from_name* being passed "BcFlather"; *read* is pointing to *bc_flather_read*; *object* is a pointer to the GtsObject associated with klass; and *fp* is a pointer to the simulation file. Function, *bc_flather_read* calls the *read* function for the GfsBcValue class (*bc_value_read*); *i.e.*, the input for the Flather BC is the same as for Dirichlet or other objects that are variations of a GfsBcClass structure, which is itself a wrapper for a GtsObjectClass structure (see file, **boundary.h**). All of the boundary conditions return a GfsBcClass pointer. The GfsBcFlather is the only one not contained in file **boundary.c**; it is located in **ocean.c** and headers are in **ocean.h**.

Using GTS Files for Input

The tides are supposed to be implementable using GTS files like with the topography. This was an original problem with using them. We have revisited this issue for the idealized domain at Santa Rosa Island using a GTS topo file that we know works. The tidal elevations are extracted from the OSU tidal model using standard methods (FILES: **gts_tides.f**, t**ide_egb.F**, **tiderot.F**)

and placed in an ***xyz*** file. This file is processed using library routines that are part of GFS using this shell script:

```
#tides.sh
lines=`wc -l k1_coefficients | awk '{print $1}'`
awk -v lines=$lines '
BEGIN {
  print lines " 0 0"
} {
  print $1 " " $2 " " $3*cos($4*3.14159265357/180.)
}' < k1_coefficients | delaunay > AK1.gts
awk -v lines=$lines '
BEGIN {
  print lines " 0 0"
} {
  print $1 " " $2 " " $3*sin($4*3.14159265357/180.)
}' < k1_coefficients | delaunay > BK1.gts
```

This produced two files that replace the macros in the previous method.

```
...
Init {} {
      A_amp = AK1.gts
      B_amp = BK1.gts
}
```

This does not work. The user-defined variables *A_amp* and *B_amp* are all zeros when viewed with gfsview3D. The gts files are ascii and can be viewed. They look very similar to those from the Gerris tutorial and from initial attempts for Mississippi Bight, which were unsuccessful.

Demonstration of Method

This is being tested by comparison with the results for Cook Strait, which works just as presented in the tutorial.

- The first step is to define a small area using the original topo file from Popinet, **bathymetry**. The domain is centered at the middle of the Cook Strait region and is 18 km across (file = **cook1.gts**). It works well using the original **coefficients** file after processing with the *tides.sh* script.
- The second step is to modify the idealized grid to fit the Cook Strait domain (174° - 174.2° longitude and -40.6° to -40.8° latitude). This is done using the original **coefficients** file (M_2 from larger model). It works but is not very smooth.
- Then, we extract K_1 tides from OSU database for this *southern hemisphere* domain and process them using *tides.sh*. The result is successful.
- The last step is to take the modified *cook1* bathymetry and make it northern hemisphere. The input files are **sri01.gts** (bathymetry), **AK1_sri01.gts** (A amplitude file), **BK1_sri01.gts** (B amplitude file). The center of the domain is: lon = 174.1° lat = 40.7°. The bathymetry file ranges from: lon = 174/174.2; lat= 40.6/40.8, and is 4 km across. The

simulation runs very slowly because Refine is 8 for lat > 29. This has been changed to *R* = 6; it runs fine and looks good (enough).

- Now, the latitude is reduced to ~30° as for Santa Rosa Island (lat range = 29.6° to 29.8°), file = **sri02.xyz**. This file is processed into **sri02.gts** with *happrox*. The K_1 tides are extracted from the OSU database with *gts_tides.f* and placed in file, **k1_coefficients_sri02**. This is then processed using *tides.sh* to produce files, **AK1_sri02.gts** and **BK1_sri02.gts**. This works but with lots of reflections because of the closed BCs on all but the bottom (south).

This last test is very close to the idealized domain that did not work. The last step is to move the longitude across the International Dateline (180°).

- Now, the longitude range is changed to 272° - 272.2°, file = **sri03.xyz**. This file is processed into **sri03.gts** with *happrox*. The K_1 tides are extracted from the OSU database with *gts_tides.f* and placed in file, **k1_coefficients_sri03**. This is then processed using *tides.sh* to produce files, **AK1_sri03.gts** and **BK1_sri03.gts**. The simulation has no values for *A_amp*, the user-defined variable read from file, **AK1_sri03.gts**. The values in file **AK1_sri03.gts** (~ 14 cm) are larger than in **AK1_sri02.gts** (~0.044 m). This is because of the location of the domain on the shelf rather than the middle Pacific Ocean.

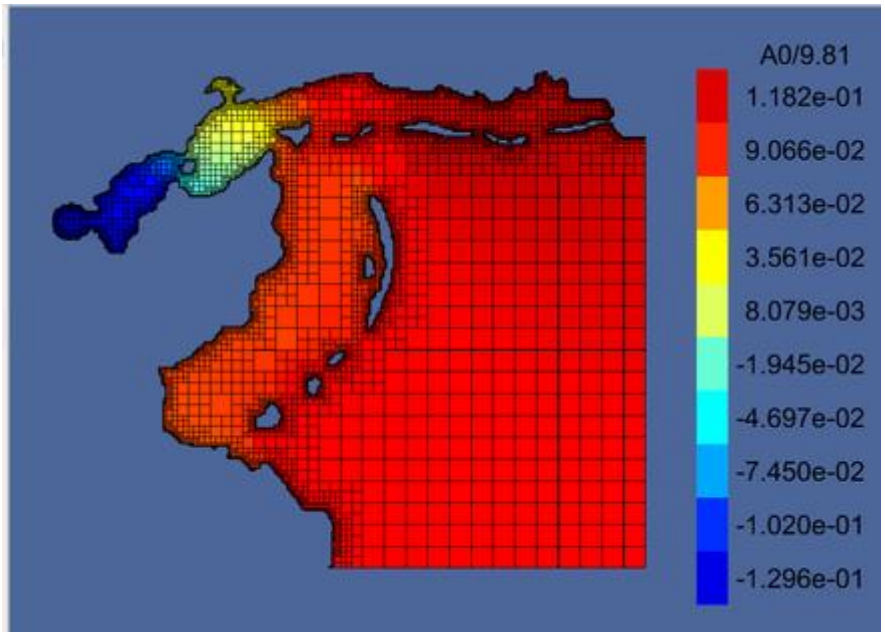
The only change from sri02 was the longitude is >180°.

- Now, we define the longitude to be negative: -88° to -88.2°, file = **sri04.xyz**. This file is processed into **sri04.gts** with *happrox*. The K_1 tides are extracted from the OSU database with *gts_tides.f* and placed in file, **k1_coefficients_sri04**, which has amplitudes and phases of ~ 0.14 m and 18°, respectively. This is then processed using *tides.sh* to produce files, **AK1_sri04.gts** and **BK1_sri04.gts**. These also have valid ranges. The results are correct.

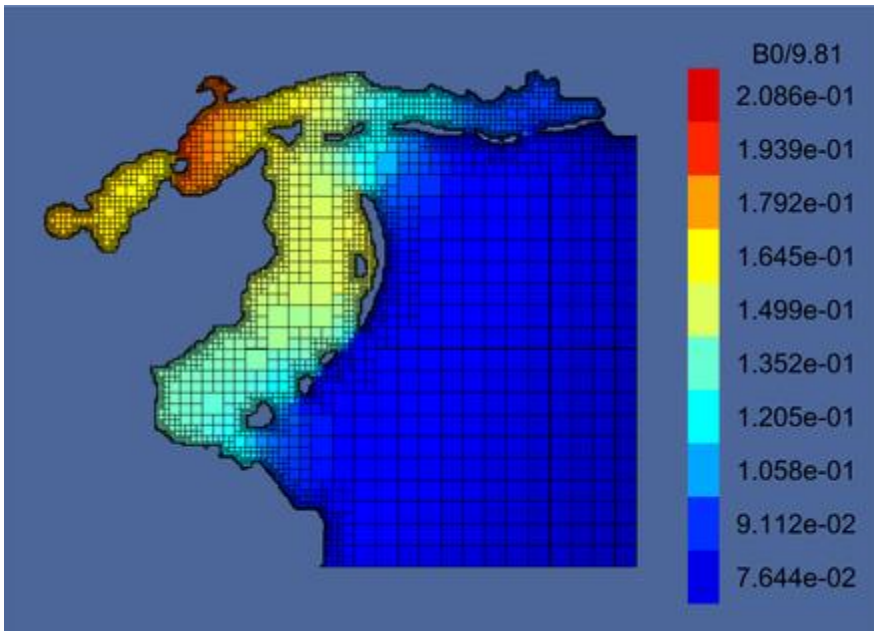
The problem is the convention for longitude. The bug appears to be within Gerris itself, as indicated by simulation *sri03*, which used the same values for the grid and the map projection (i.e., 270.1° for the center). The bathymetry and tides should always be in agreement as long as the initial **xyz** file is used to create the **coefficient** file. This is not a requirement, however; it has been tested for Mississippi Bight. This simulation (OceanModule03) uses a pre-existing bathymetry file, **bath.gts**, and new tidal files, **AK1_msb01.gts** and **BK1_msb01.gts**, as discussed in this section. The map projection uses a longitude of 270.9°.

Note that the time step will be decreased if a higher frequency output time is chosen; for example, if *OutputSimulation* is set to (step = 0.1) whereas dt is exceeding 100., the model will slow down to dt = 0.1 s. The value of A0 (see Figure below) is representative of pressure (*P*) in the model, so it must be divided by the gravity constant, 9.81. The tidal amplitude is represented by the in-phase component of the harmonic decomposition A0 (Figure 5.5A) and the out-of-phase component, B0 (Figure 5.5B).

Figure 5.5. Result after 3.47 days from OM03 using file input for K_1 tides. The units are meters.



A. In-phase component, A0.



B. Out-of-phase component, B0.

This demonstration concludes the original difficulty that was encountered in attempting to replicate the Cook Strait example for a location in a different part of the world. In this case, we were not only in the northern hemisphere, but also in the western hemisphere; New Zealand is SE quarter of globe and Gulf of Mexico is NW quadrant.

Surface Forcing with Wind

There are no good examples of how to force with a wind boundary condition. The closest example is The 3D CFD simulation (*GfsSimulation*) of air flow around a ship, the RV Tangaroa, which applies a constant wind speed of $U = 1$ (nondimensional) as a Dirichlet BC. The WaveWatch cyclone simulation uses a *GfsGlobal* function to define $U10$ and $V10$ for the wave model. These are specific to the wave model, however, and are not standard domain variables. The last is the wind-driven lake example, which uses the 2D version of the *GfsSimulation* module. This test case applies a Neumann boundary along the top with no gradient ($U = 1$).

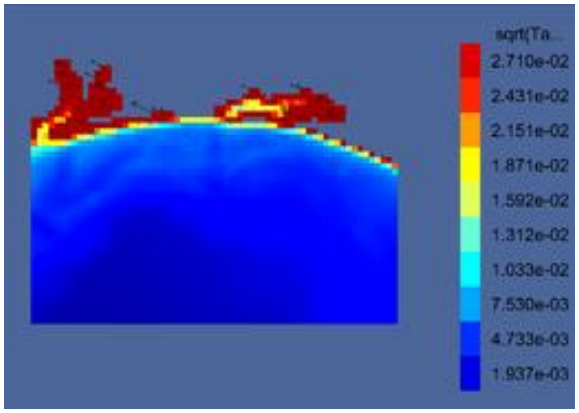
The first approach is to use the *GfsSource* class to implement $U = 0.5$, but nothing happened, not even an error. The next method was to attempt to implement a Dirichlet boundary condition on the *front* of the domain (surface of water). This was designed to implicitly assign a velocity as a function of the surface drag. Nothing happened. The next method is to use the *GfsInit* function in a similar method as the linear bottom friction.

```
Define Uwind -10.0
Define Vwind 5.0
Define CD_TOP 2.0e-3
...
Define Omega (2*M_PI/86400.)
Define Lat_Center 30.33
Define Cor (2*Omega*sin(Lat_Center*M_PI/180.))
...
SourceCoriolis  Cor
...
Init { istart = 0 istep = 1} {
     Wind = (sqrt(Uwind*Uwind + Vwind*Vwind))
     TauxW = Uwind*Wind*CD_TOP/(1000.*H)
     TauyW = Vwind*Wind*CD_TOP/(1000.*H)
     TauB = Velocity*CD_BOT/H
}
...
Init { istep = 1 } {
     U = U + dt*TauxW
     V = V + dt*TauyW
}
...
front = Boundary {}
```

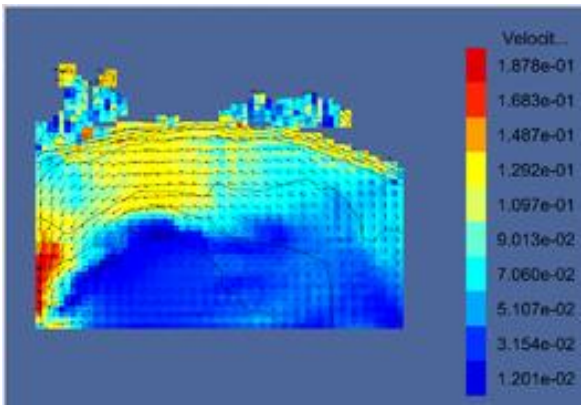
The *SourceCoriolis* function inserts the *f-plane* value directly into the equations as a term. It should be calculated from $2\omega \cdot \sin\theta$, which for ~30° N is 7.27×10^{-5}.

The wind stress is plotted in Figure 5.6A. The boundary conditions were the default, which consists of no normal flow and no-slip for parallel flow. This simulation thus produces unrealistic results for long integrations. The steady-state condition after 24 hr is useful for evaluating the overall behavior of the model. The vertically averaged currents (Figure 5.6B) show the expected increase in shallow water and the westward alongshore flow that is common to this region. The maximum current speed is 14 cm/s, which is quite reasonable. It is apparent that this simulation is becoming unreasonable, however, because of the closed eddy developing offshore. The coastal setup (Figure 5.6C) reflects the wind blowing along the axes of the estuaries. Set-down is predicted to the east while setup occurs in the west. The set-down exceeds 20 cm whereas setup is < 15 cm. The coastal setup is as expected with a > 10 cm.

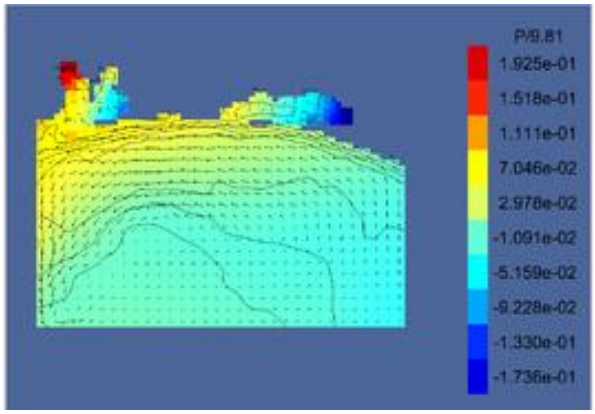
Figure 5.6. Screen dump from GfsView3D for an E-SE wind of 11.2 $m \cdot s^{-1}$.



A. Wind stress vectors plotted over contours of the wind stress magnitude.



B. Vertically average current vectors plotted over contours of the current speed at 24 hr.



C. Vertically average current vectors plotted over contours of the water anomaly at 24 hr.

Section 6: Gerris Input/Output Processing and GIS

GNU Triangulated Surface (GTS) Files

Triangulated Irregular Networks (TINs) are common objects for describing surfaces. They are used by ArcGIS, ADCIRC (and other Finite Element models), and the GTS library for input to Gerris and related software (e.g., Gfsview2D). A TIN is a vector-based model which represents geographic surfaces as contiguous non-overlapping triangles. The vertices of each triangle are known data points (x,y) with values in the third dimension (z) taken from surveys, topographic maps, or digital elevations models (DEMs). The surface of each triangle has a slope, aspect, surface area, and continuous, interpolated elevation values. The selective inclusion of points within a TIN gives the triangles their irregular pattern and reduces the amount of data storage required relative to the regularly distributed points in a DEM.

This section describes the conventions in use and explores how they can be integrated within the GAMES environment.

The format of the gts file is given in the comments for *gts_surface_write* (contained in GTS file, **surface.c**):

> All the lines beginning with #GTS_COMMENTS are ignored. The first line contains three unsigned integers separated by spaces. The first integer is the number of vertices, *nv*, the second is the number of edges, *ne* and the third is the number of faces, *nf*.
>
> Follows *nv* lines containing the *x*, *y* and *z* coordinates of the vertices. Follows *ne* lines containing the two indices (starting from one) of the vertices of each edge. Follows *nf* lines containing the three ordered indices (also starting from one) of the edges of each face.

The format described above is the least common denominator to all GTS files. Consistent with an object-oriented approach, the GTS file format is extensible. Each of the lines of the file can be extended with user-specific attributes accessible through the *read()* and *write()* virtual methods of each of the objects written (surface, vertices, edges or faces). When read with different object classes, these extra attributes are just ignored.

Details of the *GtsSurfaceClass* implementation in Gerris are discussed on the GfsSurface Main Page. The *GtsSurface* members are read from the **gts** file, as a text file (FILE *fptr). The number of vertices, edges, and faces are on the first line. The lon/lat/coefficient lines are read as vertices where the values are *z* from the format above.

These files can be created from xyz files using the following command:

```
/common/gfs/bin/happrox -f -r 1 -c 0.05 < FILE.XYZ | \
```

```
/common/gfs/bin/transform --revert > FILE.gts
```

These files can also be created using GFS tools as described in the Gerris Domain (Section 4) and Boundary Condition (Section 5) pages. Examples are given in the Applications Section (Section 11).

Output Arc Grid File

One of the simplest formats for gridded 2D data is the ArcInfo ASCII Grid format. This format can be produced by Gerris. It is simple to open with Matlab also. However, to make use of the full capabilities of ArcView with the ARCOAS Add-In, it is convenient to translate this format to a NetCDF file, which has been developed for several applications. It can also be opened by Matlab as well as Panoply (WinOS) and Ncview (Linux) applications. The ArcInfo file contains a header followed by the data on one line:

```
ncols 157
nrows 171
xllcorner -156.08749650000
yllcorner 18.870890200000
cellsize 0.00833300
0 0 1 1 1 2 3 3 5 6 8 9 12 14 18 21 25 30 35 41 47 53
59 66 73 79 86 92 97 102 106 109 112 113 113 113 111 109 106
103 98 94 89 83 78 72 67 61 56 51 46 41 37 32 29 25 22 19
etc...
```

Map Projections

In order to compare different data types, it is necessary to view them in a standard framework like ArcMap. This section discusses this question and the method used to rectify different projections.

Using the MapProjection Module

The GfsOcean module uses a Lambert Conformal grid. It is defined in the simulation file as follows. First, the size of the box enclosing the domain is defined in meters:

```
PhysicalParams { L = 205e3 }                          > 205 km across
MapProjection { lon = 270.9 lat = 30.0 angle = 0 }    > center of box
Refine 7                                    > Minimum refinement level
RefineSurface 10 combined_bath.gts    > Read bathy from file and refine
                                        to 2^10 (1024). The resulting
                                        finest resolution is 200.19 m.
```

This simulation will output the grid file at the highest resolution of 200.19 m. This should be the answer in the related files (*.asc). Instead, the answer in the associate *asc files is

```
cellsize      180.6640625000
```

This cell size may reflect the projection some how. This is not discussed in the Gerris documentation.

Projecting Arc Grid files in ArcMAP

Transforming a grid of values from one projection to another may be necessary when model output is written in a file on a grid in the projection of the model computational grid which is many times not spherical, a typically gridding requirement. Data from model output in the ARCINFO format can be read directly into ArcMAP and saved as a raster layer. The assumption is that the grid spacing is the all the same in both x and y directions in units as dictated by the projection. The procedure is described at http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#//0012000000s000000. The ASCII-to-Raster tool is available in the Conversion Tools Section in the Arc Toolbox under ToRaster.

In the Data Management Tools under Projections and Transformations is the Define Projection tool in which the user can enter in a custom projection like that of the new model data. The custom projection can be saved as a favourite for later use on other similar rasters. A raster projection file somehow appears as a .prj file, maybe through some exporting action of that layer though I have not caught it doing this yet.

To get the layers in the right place, have a world map of some sort in the TOC to establish the map display before you bring in other projected data. Presumably, the new rasters being brought in will have a reference to the already placed layer which could be in geographic coordinates or also in some projection. Layers of differing projections or coordinate systems can coexist in the same map and correctly be located with respect to each other.

Reformatting ArcInfo Grid Files to NetCDF

There are multiple methods for reformatting non-georeferenced output from Gerris into NetCDF files that can be displayed in a GIS program like ArcMap: (1) rewriting using known geographic coordinate (e.g., longitude/latitude) data; (2) rewriting using pseudo-geographic coordinates; 3) interpolation to a georeferenced grid using known coordinates values. These methods have different advantages and disadvantages. The method used is open to modification in the future. This report discusses current methods only. There are two goals to the transformation of non-georeferenced output from Gerris: (a) make it georeferenced so that it is consistent with GIS data; and (b) put it into a NetCDF file with metadata conforming to the COARDS standard.

The ArcInfo grid output is set in the simulation file using the *OutputGRD* keyword. The OutputGRD class can write multiple fields to one grid file by not indicating a time in the simulation file; for example, *p-%g.asc* indicates that only one field should be written to the file. If no format specifier is given, multiple times will be placed in the same file. This has a potential advantage for processing the fields into NetCDF files because ARCOAS can read time

series of variables from multiple NetCDF files. If separate **grid** files are used, they can be joined as follows:

```
[keen@typhoon NS-6]$ cat w-*.asc >> w_all.asc
```

This will append all of the **w** files to the **w_all.asc** file, assuming the dates sort correctly; otherwise, it must be done manually.

Gerris has an error in the printing function that does not place a carriage return "\n" before each new field (i.e., "ncols"). Consequently, the header (see above) begins on the same line as the preceding data field and is not read properly. The first step, therefore, is to edit these multiple-field files and insert a carriage return before every occurrence but the first of "ncols".

Reformatting to COARDS without Georeferencing

The ArcInfo grid file contains geographic coordinate data that is not gridded because of the Lambert Conformal projection. If the region is small enough, the errors can be acceptable. This translation of an ArcInfo Grid file to a NetCDF file is accomplished using a perl script. The attributes of the variables can be made to conform to the COARDS File Format when writing the NetCDF files. The original script was written to translate files written by FORTRAN programs, which are indexed from the bottom up. An option was added to read files from the top down. This script is:

```
/home/keen/common/ascii2coards/ascii2coards.pl.
```

The script is used as follows:

```
Usage: ./ascii2coards.pl [-d dFile -i inFile -o outFile] | [-h]

  -d dFile   : file of descriptors (describes the data in inFile,
               ASCII format)
  -i inFile  : file of block data (ASCII format)
  -o outFile : name of NetCDF file to be created
  -h         : what you see here.
```

The *ascii2coards* script can be invoked for each variable/file and it is added to NetCDF files to fit any desired format. The resulting NetCDF files can be made COARDS compliant.

Output from the 2D Vertical CFD Model

The ArcInfo grid file is used for the 2D vertical CFD model in addition to the *GfsOcean* and *GfsRiver* modules. There are no geographic coordinates in these files; both axes are from -0.5 to 0.5 (plus any defined translations). The **ascii2coards** descriptor file contains variables that control the tranformation to pseudo-geographic coordinates for display in a GIS program. This is unnecessary for plotting with Matlab or other general programs.

This example is for the output from the 2DV CFD model from the Tamar River (Keen et al., 2013b, MR NRL 7300, *Hydrodynamics and Finne-Grained-Sediment Dynamics in the Estuary Turbidity Maximim*). In order for these fields to display properly in ArcView (using ARCOAS) and Panoply, they must use a 2DH convention. This requires using the COARDS dimensions, *longitude* and *latitude*, where *latitude* is the name for the *vertical* axis. This is not a straightforward process; however, because of an error in the writing of the **.grd** file by Gerris.

```
Parm = 'water_v'
longName = 'Vertical Velocity'
varType = 'float'
units = m/s
fillValue = -9
missing_value = -1.e+34
dataMultiplier = 1.
scale_factor = 1.
dataAddend = 0.
NWstartFlag = T
headerCount = 6
xAxisName  = 'longitude'
yAxisName  = 'latitude'
zAxisName  = 'depth'
tAxisName  = 'time'
xAxisPars  = 'longitude'
yAxisParm  = 'latitude'
zAxisParm  = 'depth'
tAxisParm  = 'time'
xAxisLongName  = 'Distance along channel'
yAxisLongName  = 'Distance from channel center'
yAxisLongName  = 'Height'
tAxisLongName  = 'Time Step'
xUnits = 'Meters'
yUnits = 'Meters'
zUnits = 'metres'
tUnits = Minutes
xOrigin = -4.215455
yOrigin = 50.497538
zOrigin = 0.
tOrigin = 0.
tOriginDate = 1998-09-16 15:48:00
xIncrement = 0.00563
yIncrement = 0.00563
zIncrement = -0.0625
tIncrement = 1.
xStart = 0
yStart = 0
zStart = 0
tStart = 0
xCount = 768
yCount = 64
zCount = 1
tCount = 6
```

Figure 6.1 shows how such a result looks when put into pseudo-geographic coordinates in *Panoply*. The upper left corner is placed on the world map where it is indexed; this can serve as

a technique for placing the image geographically. These files are easily processed using ARCOAS tools.

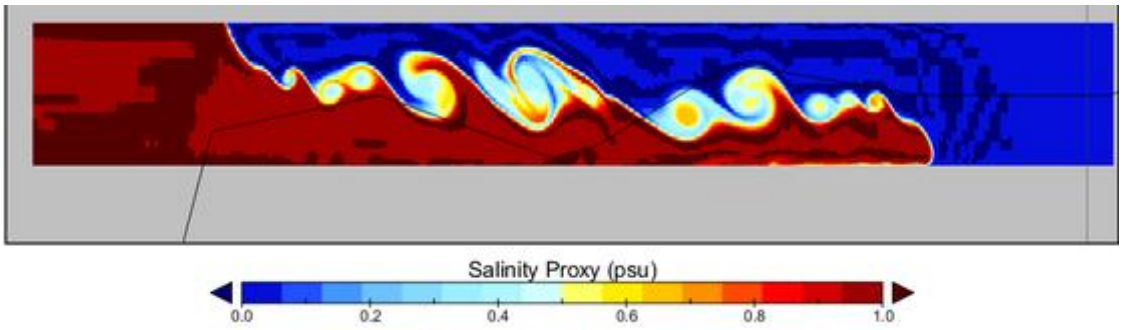


Figure 6.1. Example of output from Gerris transformed to lat/lon coordinates using ascii2coards.pl

Output from the *GfsOcean* Module

A more traditional grid orientation that can be processed into a NetCDF file is the 2DH grid used for the tidal study (Gerris Mississippi Bight Tides). This is a more conventional orientation that utilizes geographic coordinates in the ArcInfo Grid file. The domain in Gerris is specified by the latitude and longitude of the center and the total size of the box. This is done because there is no grid *a priori*. We will use an example with the following domain specification:

```
PhysicalParams { L = 185e3 }
MapProjection { lon = 270.9 lat = 30.2 angle = 0 }
```

The headers from the **p-200000.asc** file are:

```
ncols           911
ncows           628
xllcorner       -72174.060938
yllcorner       -92469.800000
cellsize        180.6640625000
nodata_value    -9999
```

The approximate length of an output cell along the x axis is $(360°) \cdot (180.6640625000)/[\cos(30°) \cdot (40008 \times 10^3)] = 0.001806279°$. The lower left corner would then be $270.9° - 0.001806279° \cdot 455 = 270.081°$. The y axis would not be corrected for latitude; $30.2° - 0.001625° \cdot 314 = 29.68975°$.

Two additional parameters have been added to the ascii2coards.pl script to assist in processing the Gerris ouput from the Arc Grid files: (1) transform, which is a simple multiplicative coefficient for data; and (2) badvalue, which is set to the missing/fill values from the ascii file. This is useful to maintain these flags for later processing. These are assigned in the ascii2coards*.in file. The default values are 1 and -9999, respectively, for transform and badvalue.

The *ascii2coards* Perl script can be used to write a *COARDS* compatible NetCDF file from the grd file but it is not georeferenced because the cell size in the grid header is constant, which is not the case for the Lambert Conformal projection used in Gerris. The resultant raster file displayed in Arc (Figure 6.2) does not match the satellite images. This can be seen in the western part of the domain in the image below.

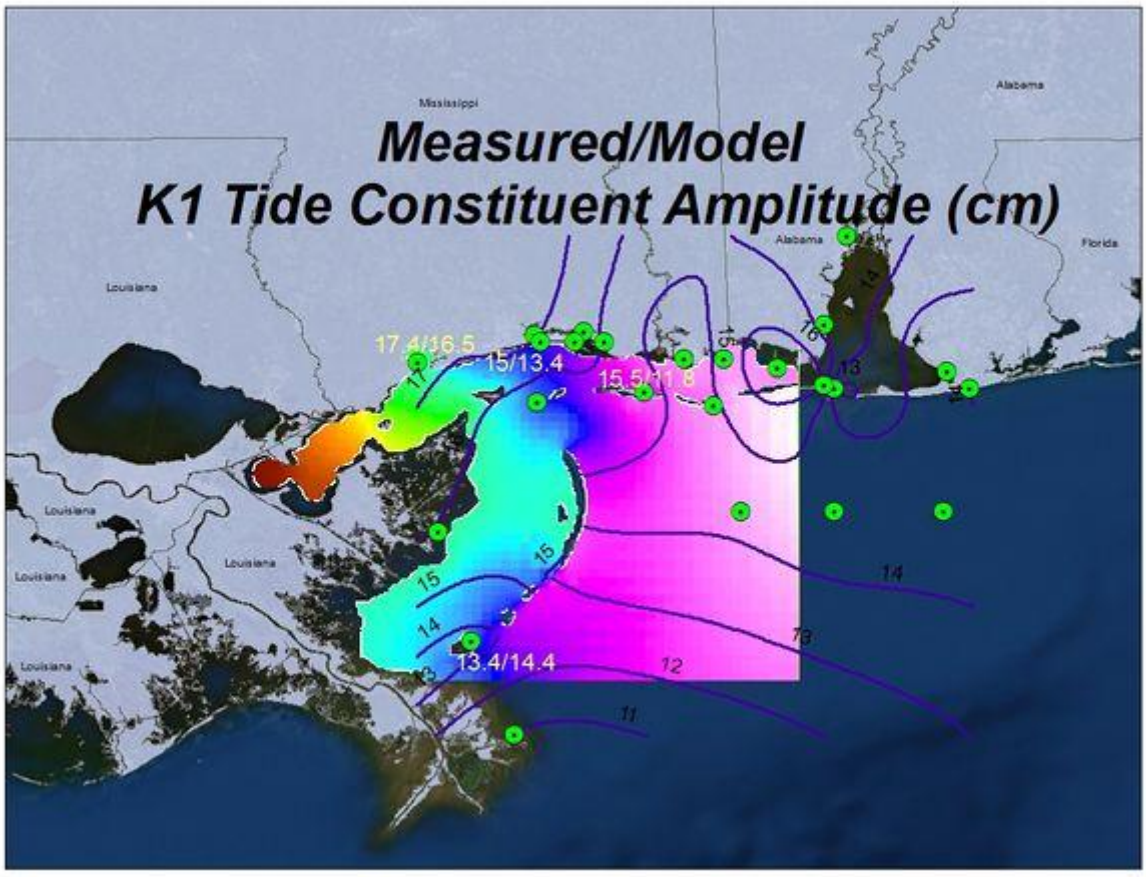


Figure 6.2. Screen dump of ArcMap view of K_1 amplitudes (cm) from *GfsOcean* module after processing by the *ascii2coards* script; and the observed/modeled values at selected points.

Creating Georeferenced NetCDF Files

This section describes two methods for producing accurately georeferenced data sets that can be plotted by any GIS-based visualization program. The first uses ArcMap interpolation functions and will work on unstructured data with (potentially) coastline data used as a barrier to produce the file or mask the resultant. The second uses ArcMAP projection functions that only work on structured data sets.

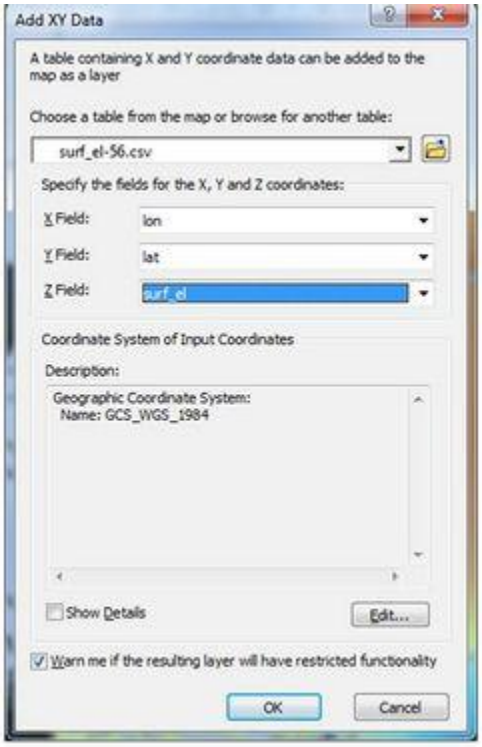
Projecting to Georeferenced Coordinates with ArcMap

Gerris outputs an ASCII file in the Arc Info Grid format. This is not a georeferenced file, however, and it must be processed to be accurately represented on the Earth's surface using ArcMap. This section discusses methods for completing this. The projection used by Gerris is the Lambert Conformal Conic, which is included in the *proj* library. The first step is to output the desired model variable (e.g., pressure) as well as the *x* and *y* coordinates. For the ocean module these are longitude and latitude, respectively. The model output is written at the finest refinement and the *grid* is extrapolated outside of the Gerris domain. The resulting files indicate the *ghost* grid points as special values (-9999). This file can be opened by Arc but it is not georeferenced because of the special values for non-grid points. It is thus imported as a **point file**.

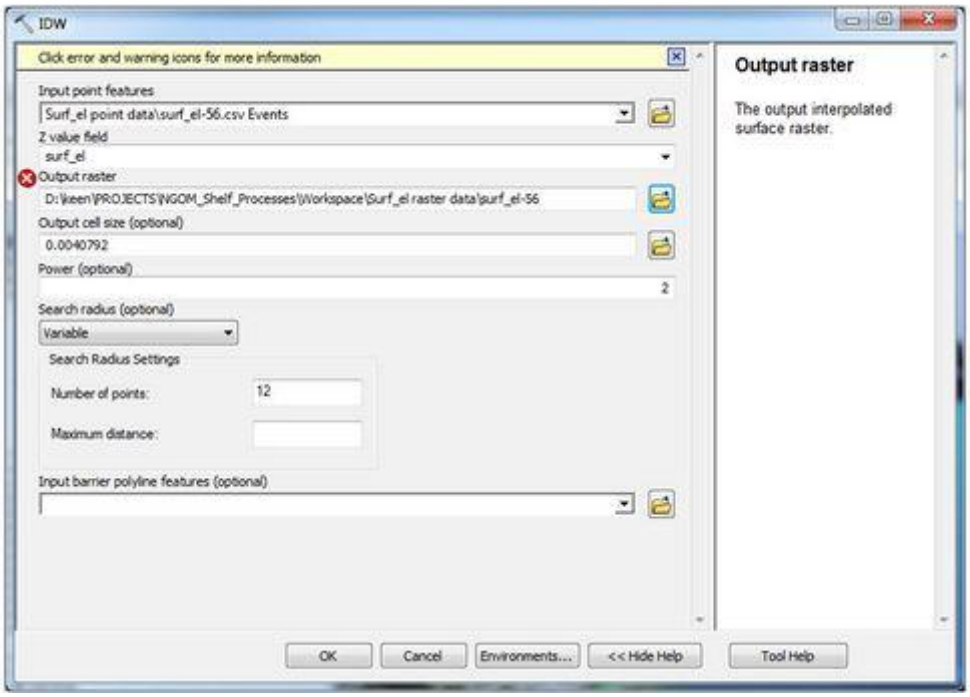
Georeferenced layers can be created from the grid (ASCII) files output from Gerris Grid files using the following method. The ASCII files are easily read by Matlab (*remove_nan_from_grd.m*) using the *fgetl* function to read the header and data lines as strings that are parsed using *sscanf*. Each grid file contains only one variable and time, because of a bug in Gerris that fails to place a carriage return (\n) at the end of a data line; thus, if multiple times are written, they are concatenated in a ridiculously long line that must be deciphered with substantial trial and error. The script writes only valid data values to a file (e.g., **surf_el-XXX.csv**) with one longitude, latitude, and data value (times a conversion factor) on each line with separating commas. This script also finds the global minimum and maximum for all of the data. This is used to restrict the data range that is plotted by ArcMap.

The **csv** files are imported into ArcMap using the *File.Add_Data.Add_XY_Data* function (Figure 6.3A). It is convenient to collect these into a *Group Layer* as seen in the Table of Contents from the Screen Dump. The data are displayed as points (Figure 6.4) that are georeferenced. The ArcToolbox is used to run the necessary functions to transform these point data into a usable format like a NetCDF file. This Toolbox is represented as a tiny red tool box on the tool menu. It is very difficult to see. For this project we selected the IDW (Inverse Distance Weighted) interpolation method to make a raster layer of these points. This function is found as *ArcToolbox.Spatial_Analyst_Tools.Interpolation.IDW*. Selecting this tool opens a dialog box (Figure 6.3B), which allows the desired point file to be selected.

Figure 6.3. ArcMap menus used for entering and interpolating point data.



A. Menu for adding the Gerris *grd files to the current ArcMap Document.



B. IDW dialog box used to generate a raster object from a set of points.

The point data (Figure 6.4) are accurately located and georeferenced but they cannot be used for advanced processing because they are not raster data. They must be interpolated to a specified grid using a method like inverse-distance-weighted (IDW) (Figure 6.3B). A barrier can be used to constrain the resulting contouring by features like islands. The resulting raster objects (Figure 6.5A) from surface elevation output from Gerris are stored in the *Surf_el_raster data* Group Layer. These are not files, but they appear as directories in the file system; for example, on Tornado they are found in D:\keen\PROJECTS\NGOM_Shelf_Processes\Workspace as a set of directories (e.g., surf_el-59). They are not intended for user access.

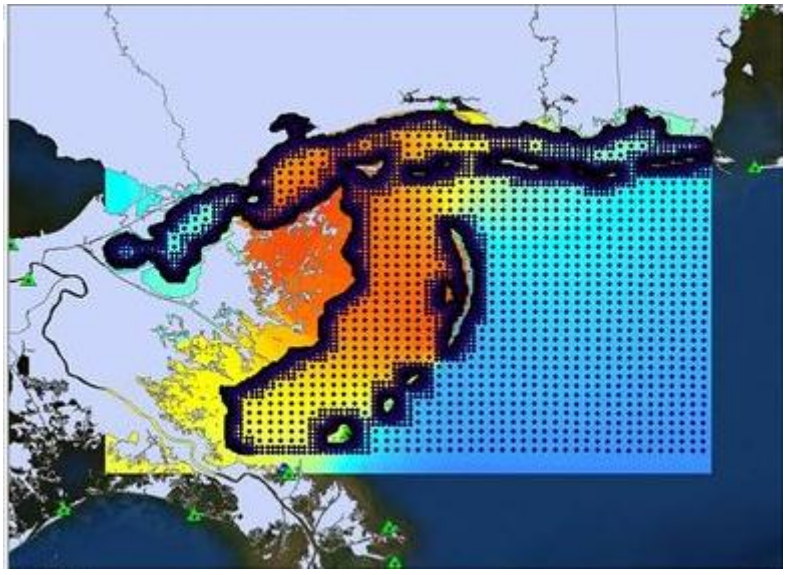
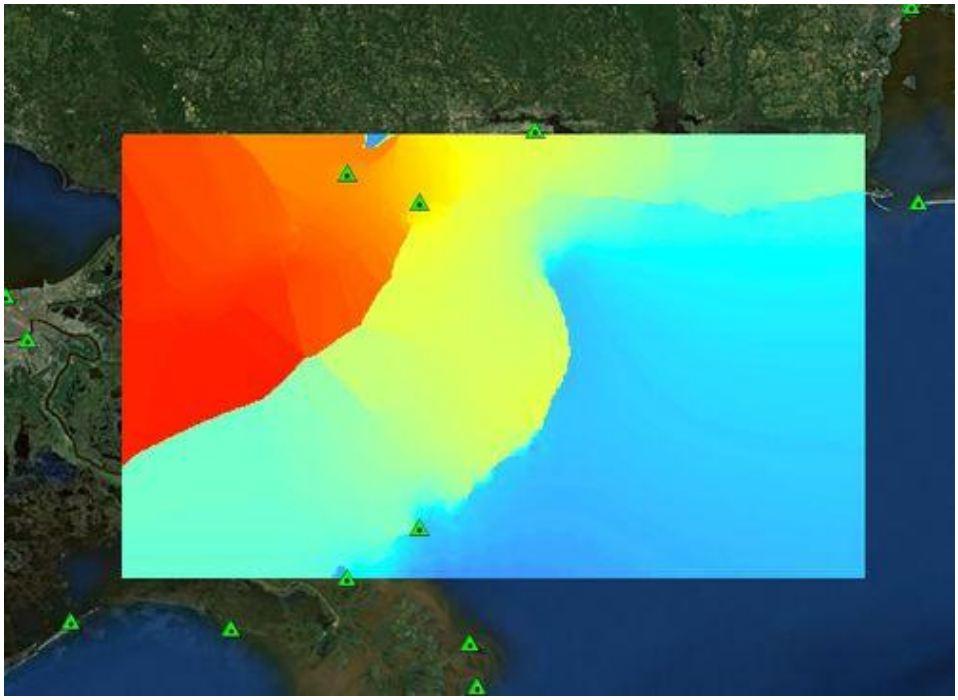


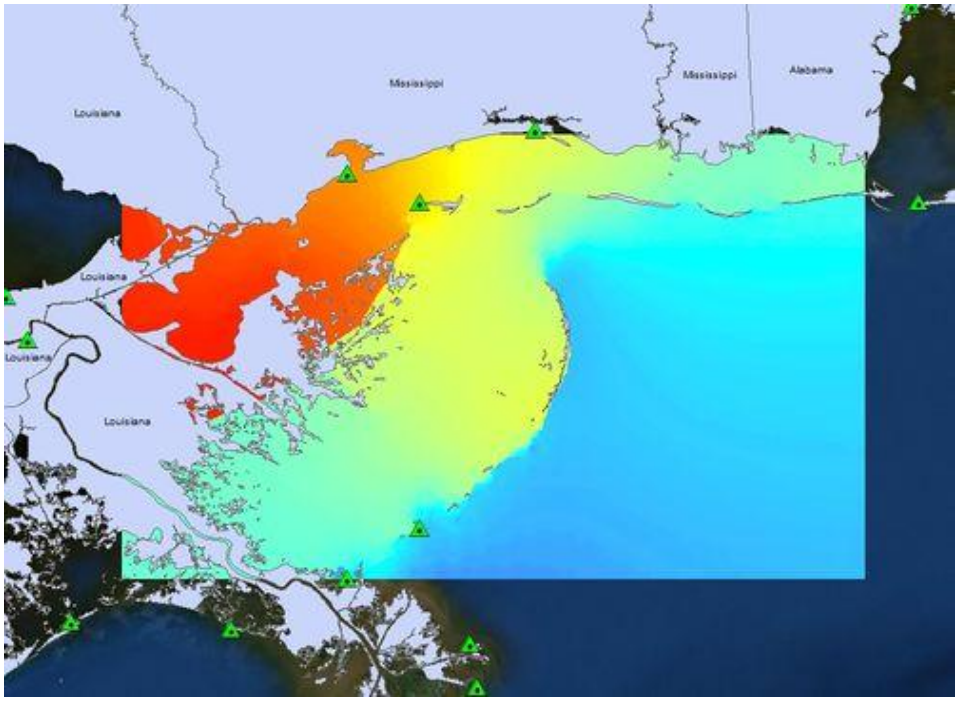
Figure 6.4. Screen dump of one of the converted Grd files as point data in ArcMap.

These raster objects are exported as Arc NetCDF files using the *ArcToolbox.Multidimension_Tools.Raster_to_NetCDF* function from the tool box. This dialog box is very similar to the IDW box but it allows selecting the raster object from either the file system or the Table of Contents. The output location should be selected to fit into the data structure; however, this is not a permanent file. These files are placed in the D:\keen\PROJECTS\NGOM_Shelf_Processes\Workspace\msb_refine10 folder (**surf_el-56.nc**, etc). They are not included in the Table of Contents for the current ArcMap Document, however.

Figure 6.5. Screen dumps from ArcMap raster output.



A. Data view map of Gerris output at τ = 27 hours, showing the effect of IDW interpolation without a barrier.



B. Data view map of Gerris output at τ = 27 hours, showing the effect of a mask IDW interpolation without a barrier.

These exported NetCDF files plot correctly in either *ncview* (Linux) or *Panoply* (WinOS) but they do not contain the required attributes to meet the *COARDS* standard.

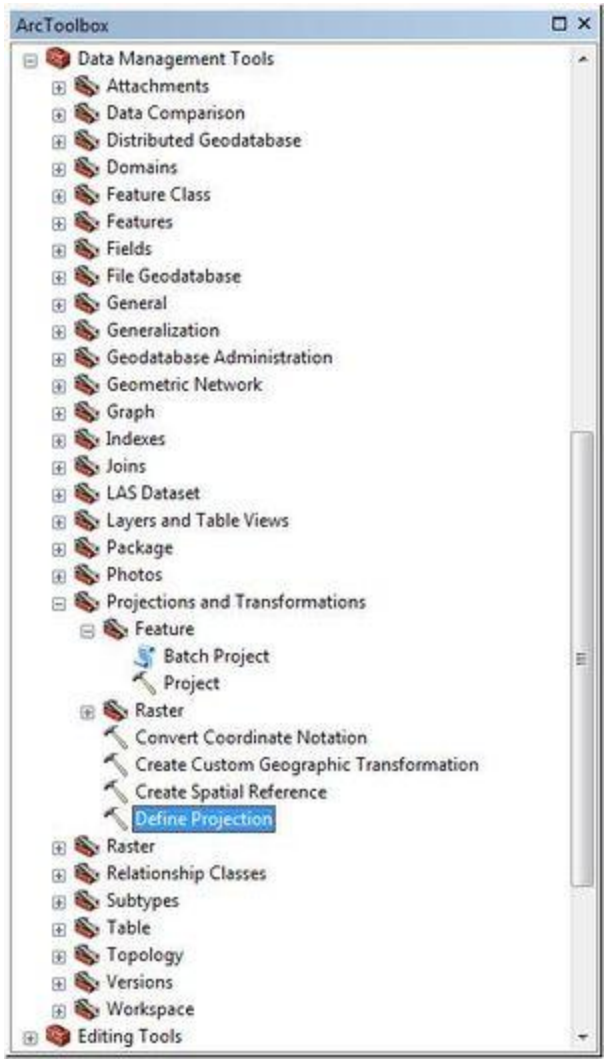
The new NetCDF raster layers are listed in the Table of Contents under the *coards_netcdf* Group Layer. These are no different than the *raw* NetCDF files or *Surf_el raster data* groups except for having caps on data and extra attributes not used by ArcMap. All of these raster objects have data extending throughout the extent of the map because of the use of IDW with no barriers. This is masked by the *World_Administrative_Divisions* data base from Arc (Figure 6.5B). The mask has no impact on the data, however; it is only for visual reference. This surface still contains values that may be invalid. The *spline* produced more bad values in the delta area, which made the IDW more useful for this study.

Georeferencing using ArcMap Projection Tools

ASCII GRID files with the appropriate header as described in the ESRI Help can simply be dragged and dropped into the map view (or you can got through the Add Layer menu) and displayed. Until the projection for the data is defined, the header has no meaning and the image of the data will be incorrectly placed geographically.

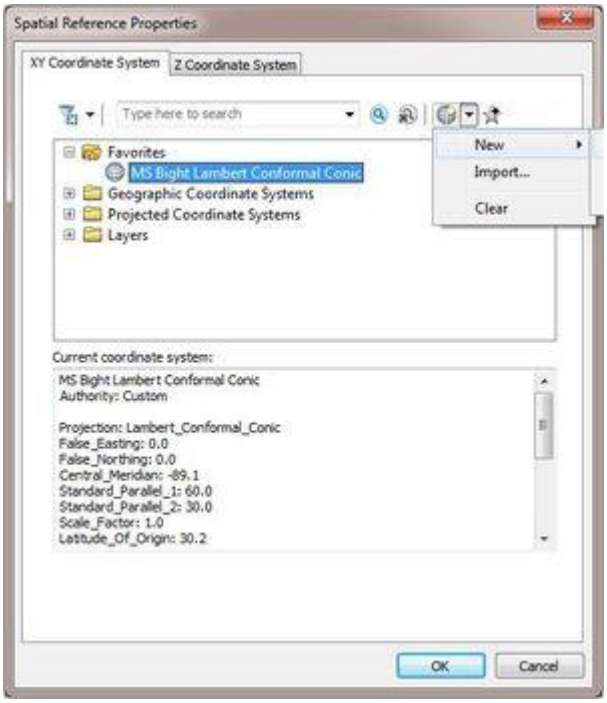
To add a projection to the resulting raster layer, select the *Define Projection* tool (Figure 6.6A) in the Data Management Tools toolbox (Figure 6.6A).

Figure 6.6. ArcMAP Menus for Define Projection.



A. Tool box menu.

B.Toolbox Selection Define Projection.



C. Spatial Reference Properties selected in tool to Define Projection.

Go to the ESRI Help page for more information on using the Define Projection tool.

Gerris uses a Lambert Conformal Conic grid projection, for which the two critical pieces of information are: (1) the *Central_Meridian*; and (2) the *Latitude_of_Origin*. The file is p-1.01e+06.asc was transformed using the following parameters (Figure 6.7) in the properties menu (Figure 6.6C):

```
False_Easting: 0.0
False_Northing: 0.0
Central_Meridian: -89.1
Standard_Parallel_1: 60.0
Standard_Parallel_2: 30.0
Latitude_Of_Origin: 30.2
Linear Unit: Meter (1.0)
```

The central Meridian and Latitude of Origin are taken from the *MapProjection* line in the Gerris simulation file (see above). Save your work to an MXD file and both the project file and raster files will retain the projection information. Additional files are created in the process: **.prj** and **.xml** with the same root names as the original file. Compare this georeferenced data to the non-georeferenced data in Figure 6.7.

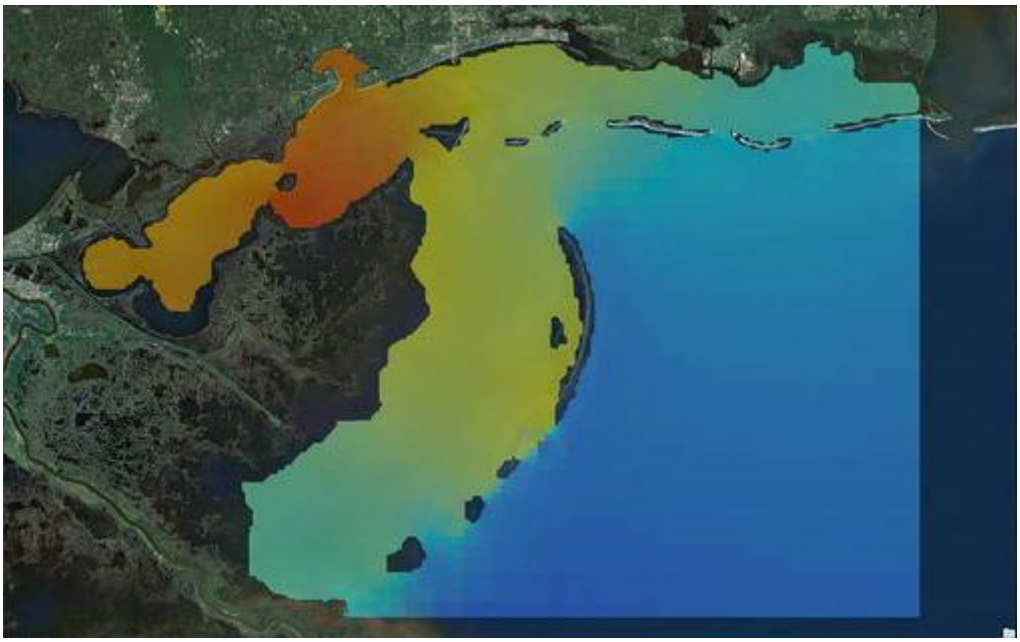


Figure 6.7. Transformed data from the Ocean module displayed in ArcMap.

Creating a COARDS-compatible NetCDF File

The NetCDF files from the ArcMAP methods described in the previous section are georeferenced but they are not COARDS compatible. Thus, they cannot take advantage of ARCOAS NetCDF database functionality.

In order to make them fully compliant, they are opened by a Matlab script (*make_coards_compliant.m*), which adds attributes, makes a final conversion for desired units, and caps the range to aid in visualization. The input NetCDF files are named following the *COARDS* standard: [model name]_[region]_[cycle date]_t[forecast hour], where: cycle date refers to the time when the forecast model was reinitialized with a new initial condition generated by (typically) some form of data assimilation; forecast hour is the number of hours

since the initialization (referred to as *tau*). It is noteworthy that no reversing of the *y* axis is necessary (the usual Fortran to C problem) if the files are to be imported into ArcMap using the *ArcToolbox.Multidimension_Tools.Make_NetCDF_Raster_Layer* function. If the alternative is used (*ARCOAS.Add_Layer_Tools.Add_layer*), the ARCOAS functionality may be available but this necessitates reversing the *y* axis because external libraries are used to import the NetCDF files through ARCOAS. These libraries have undocumented differences that cause this problem.

Section 7: Testing CFD Solvers

Two-Dimensional CFD Testing

This section describes efforts to explore the capabilities and implementation issues related to using Gerris for both micro-scale and field-scale problems. The CFD engine is used for the small scale whereas there are several modules implemented in GFS for ocean problems. One of the first tasks that Gerris is being used for is to simulate tidal flow in an estuary. This work was actually undertaken prior to the. One of the conclusions from this work was that it is better for geophysical flows to use dimensions in setting up the model. However, this does not preclude the need to have a solid understanding of the use of non-dimensional models because this approach is the norm in computational fluid dynamics. There is a discussion of some of the issues related to this approach in this section.

This page is the main location to discuss tests intended to explore the operation and accuracy of the Gnu Flow Solver (GFS or Gerris). Two tests have been undertaken thus far: (1) the lock-exchange problem; and (2) the incorporation of tidal constituents in the Mississippi Bight. Any attempt to implement a new theory or technology is expected to be difficult. Gerris and CFD are no exception.

Gerris is a very robust code and it comes with a good data visualizer that allows the adaptive grid output to be analyzed accurately. This viewer is called gfsview2D or gfsview3D, depending on the dimensions of the simulation to be viewed. Gerris also has a built-in function to print ***ppm** files. These files are fast to visualize and can be examined while the program is still running. They are viewed using the *animate* program (Linux). In addition, an ESRI grid ASCII file can be produced for 2D scalar fields. These files output the results at the highest refinement in use at the time the output is generated. They are easily plotted using Matlab or imported into a GIS application. The results presented in this report use these three programs. Screen dumps are used for the gfsview2D and animate results and EPS files from Matlab. These files are translated to JPEG for the wiki. Arc has not been used to date.

The starting point for applying Gerris is the page on the Gerris web site (http://gerris.dalembert.upmc.fr/gerris/examples/examples/index.html). These can typically be reproduced with very little difficulty. This is useful to develop some level of competence with the code. They are also useful as templates to develop new problem simulations. The web page has a lot of documentation and is evolving continuously but there are still some limitations with respect to accessing Gerris variables and how to implement user-defined variables using macros.

This page describes some simple experiments that were completed with Gerris prior to the lock-exchange tests. The goal is simple; can the 2DV CFD model be used to simulate tidal flow with density variations and associated sedimentation processes? To address this objective, we have examined the use of an oscillatory boundary layer at the downstream end of the domain. This boundary is further examined with respect to a prescribed logarithmic profile for the boundary

condition. We also investigated the use of bottom friction to develop a logarithm profile within the channel. This flow was then examined in combination with a equation of state (EOS) for a tracer that represents salt. The final simulation uses a volume of fluid (VOF) model that is part of Gerris.

These simulations began by following principle (4) for using Gerris. The closest example from the Gerris web page is Example 2.1, the Benard-von Karman vortex street. They were all completed without dimensions.

Dimensionless Scaling

CFD codes like Gerris are often nondimensional. I have noted this in several papers. This requires a set of characteristic dimensions and other fundamental properties that are used to scale the input and output. There is a good discussion of [scaling parameters] online. An important relationship is that Re_M = Re_R, where Re = $U\_C \cdot L\_C/\nu$ is the Reynolds number; U_C = a reference velocity, L_C = a reference length, and ν = the kinematic viscosity. The M and R subscripts refer to the model and real properties, respectively, but we can simplify this further by using upper case letters for real variables and lower case for model variables.

The Gerris users guide states that, by default, the density is 1 and molecular viscosity (μ) is zero. This means that there is no explicit viscous term in the momentum equation. The size of the unit GfsBox is 1. It is furthermore suggested that all physical input parameters be scaled by a reference length (the physical length of the GfsBox). One example that is discussed is for a ship 150 m in length and a wind speed of 50 m/s. The unit GfsBox can be 450 m (i.e., L) so that the ship model must be scaled by 1/450. This is required to transform the ship to a length relative to 1, which is the nondimensionalized unit GfsBox. To interpret the results in terms of physical units, is is necessary to multiply the length output by 450. Analogously, the wind speed can be nondimensionalized using the maximum wind speed of 50 m/s (i.e., divide by U = 50). Velocities are rescaled on output by multiplying by 50. The reference time $T = L/U$ = 9 s for this problem. We multiply both t and dt by T = 9 s for output.

As long as we have not set a physical parameter (GfsPhysicalParams), the input is implicitly as follows: l = 1 m, u = 1 m/s, and t = 1 s. The density is 1 kg/m^3 and the dynamic viscosity μ is 0 (kg/m/s). The dynamic viscosity is a characteristic property of all liquids, but the input value can be used as a tuning parameter to produce a flow with the desired value of Re.

We can apply this methodology to the tidal flow from this section. We can find the appropriate kinematic viscosity υ to use in Gerris.

$$\upsilon = u \cdot \upsilon / U \cdot l / L$$

$$= u \cdot \upsilon / U \cdot \text{ scaling factor}$$

Using the peak tidal current and water depths for the model and real cases. The mid-depth current varies continuously over the wave period of 15 steps. One value of U that we can use in our scaling is 0.7 m/s, which is the approximate maximum flood tide current speed. It shouldn't

matter what reference velocity we use as long as we are consistent in applying it. For this problem, we want the characteristic length to be the approximate depth of the Tamar River at high tide, 4 m.

$$\upsilon = (1\ \text{m/s}) \cdot (1.006\times10^{-6}\ \text{m}^2/\text{s}) / (0.7\ \text{m/s}) \cdot (1\ \text{m}) / (4\ \text{m})$$
$$= 3.57\times10^{-7}$$

We want to verify our result using dimensions. The channel is nine times as long as it is deep (i.e., 9 GfsBoxes end-to-end), which represents 36 m. The oscillation period is 15 steps and the entire simulation is 15 steps. We expect this simulation to end with the wave front where it began because we set the maximum *model time* to be the oscillation period. We can check the *real time* by:

$$T = t \cdot L / 1 / U$$
$$= 15 \cdot (4\ \text{m}) / 1 / (0.7\ \text{m/s}) = 85.7\ \text{s}$$

In other words, our oscillation period t is ~86 s and dt = 5.7 s. The model velocity referred to by u is a reference velocity as is the real velocity U. We can refer to arbitrary velocities with lower case subscripts: The wave current at some time would then be given by:

$$U_r = u_m / u \cdot U$$

For example: if u_m = 0.4, U_r = (0.4) / (1) · (0.7) = 0.28 m/s.

The *average* velocity of the wave front can be estimated from the tracer plot: the front reaches its maximum extent (4.46 GfsBoxes/~18 m) in < 8 steps (~43 s). The mean velocity is thus 0.42 m/s. The mean of a sine curve for the interval (0 to л/2) is (л/2)-1 or 0.64. The mean velocity in real dimensions is thus consistent with the theoretical value (0.64·0.7 m/s = 0.45 m/s). This alternate value of U demonstrates a potential problem with using dimensionless analysis for time-dependent problems.

In order to interpret the result for a tidal period of 12 h, we must rescale the characteristic length L and reference time T of the real flow. We can estimate L_R from u_R using the desired simulation length (12 hr) and number of GfsBoxes (9):

$$L = (0.42\ \text{m/s}) \cdot (43{,}200\ \text{s}) / (9\ \text{boxes}) = 2016\ \text{m}$$

This simulation represents a tidal flow with a mean current of 42 cm/s in a channel 2 km deep and ~18 km long. The mean velocity for this "larger" problem is the same as for the "smaller" problem. The reference time $T = L / U$ = (2016 m)/(0.42 m/s) = 4800 s. The model time step, $dt = T/t$ = (9·4800 s) / (15) = 2880 s. The rescaled model viscosity, υ = (1 m/s)·(1.006×10^{-6} m^2/s)/(0.42 m/s)·(1 m)/(2016 m) ~ 10^{-9}.

Oscillatory flow

These simulations used a 12 hr tidal period and a prescribed logarithmic boundary condition given by: $U = (u_*/K)\cdot\mathrm{LOG}(z/z_0)$ where: $u_* = 0.04$ m/s; $K = 0.4$; and $z_0 = 0.001$ m. The *bottom* was frictionless (free-slip). The surface velocity is 0.7 m/s and the mean is 0.5 m/s by default. This is very close to that during a flood tide in the Tamar River. This is implemented as a Dirichlet BC on the left side of GfsBox 1.

```
GfsBox {
   left = Boundary {
          BcDirichlet U ( (1. * sin(2.0*M_PI*t/(15.0))) * \
                          (0.04/0.4)*log((y+0.5)/.001))
          BcDirichlet T ( 1. )
   }
}
```

The tracer is transported by the current, with limited mixing because there is no vorticity. The dynamic viscosity (μ) is 0.0078125 kg/m/s. The Re = $U*L/\nu$, where: $\nu = \mu/\rho$; $U = 0.5$ m/s; and $L = U*T$. We estimate T from the model input and results; for example, the simulation length of 15 steps represents 12 hours. The tidal front propagates a total of 5.9 GfsBoxes in 12 hr. Thus, the *average* or characteristic time scale T to transit a box is (5.9)/12 ~ 1/2 hour or 1800 s. Consequently, U is 900 m and Re ~ 2.3×10^8. We note that $\rho = 1000$ kg/m^3.

The flow is smooth (Figure 7.1) despite the large Re because there is no density difference and the frictionless bottom generates no turbulence.

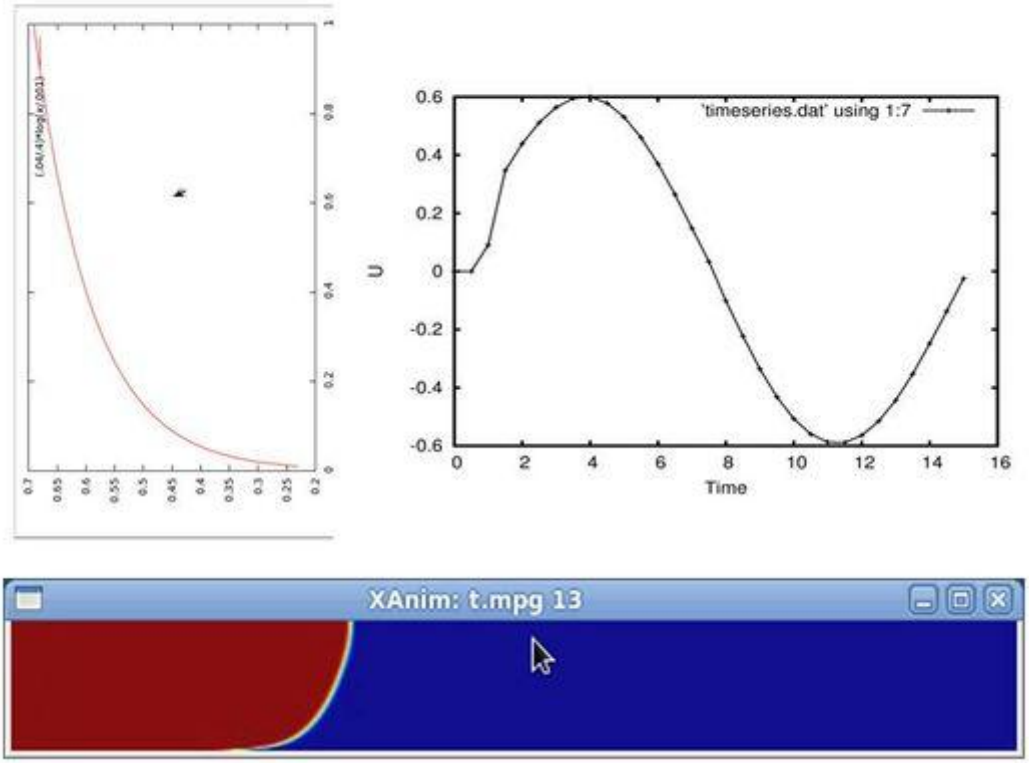


Figure 7.1 Tidal inflow boundary condition. (A) profile of nondimensional current; (B) nondimensional current at mid-depth at nondimensional time; (C) conservative tracer after 13 nondimensional time steps.

We note that the tracer profile at $t = 13$ is a logarithmic profile (Figure 7.1C) that matches the inflow profile. The period of the oscillation is 15 time units, so the flow is to the left. This doesn't make sense until we remember that the bottom is free slip. If the profile remains logarithmic within the domain, it should have reversed and the final profile should be vertical as it was at the beginning. This didn't happen because the model adjusted for the inflow and produced a uniform profile within a few boxes of the right boundary. This was verified.

Methods that can be used to create a logarithmic profile within the domain will be addressed in the next section.

Logarithmic current profile

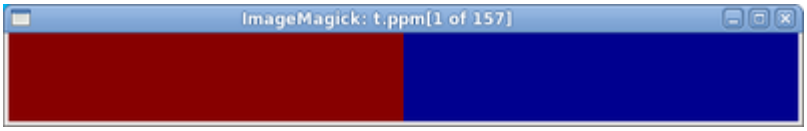
The flow within a channel naturally develops a logarithmic profile in response to bottom friction. We were not able to maintain such a profile with a boundary condition at the inflow end of the channel. This can be approached in two ways for Gerris, which does not include a parametrization for subgridscale processes. If we maintain a high resolution at the boundary (i.e., a DNS problem), we can supply roughness elements in the flow that will create vorticity. These eddies will propagate within the flow as they are dissipated by the viscosity. We can also

implement an eddy viscosity model like Mellor-Yamada or *k-Epsilon*. This would allow us to solve the Reynolds-Averaged N-S equations (RANS).

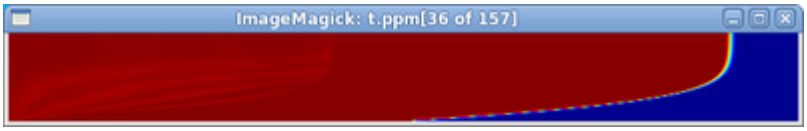
The first type of simulation we can examine is a no-slip bottom with viscosity. This simulation uses a uniform inflow of 0.7 m/s, a dynamic viscosity of 0.0078125 kg/m/s, and Pr = 10. The mixing is revealed by an initial vertical tracer front at x = 4 (~ mid-length), which rapidly becomes logarithmic. The channel must be longer than the maximum excursion length of the front. Otherwise the *OutflowBoundary* at the right end will cause a stratified distribution when the front re-enters the channel. The length of the domain was therefore increased to 9 boxes. The conditions are otherwise the same as for the previous example but the simulation goes for 30 steps (two tidal cycles).

This tracer pattern over two tidal cycles is shown in Figure 7.2. The current profile is logarithmic well before it reaches the front, which then forms a characteristic profile as well. This sequence of images shows that the logarithmic current profile does not generate any mixing and the tracer always returns to almost the same distribution at any given point in the tidal cycle after the initial flood tide. Mixing is occurring but slowly.

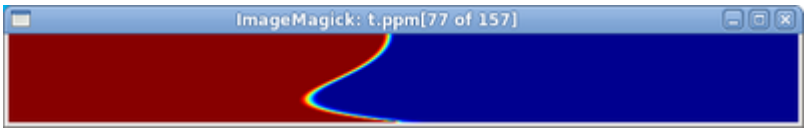
Figure 7.2. Time evolution of tracer distribution.



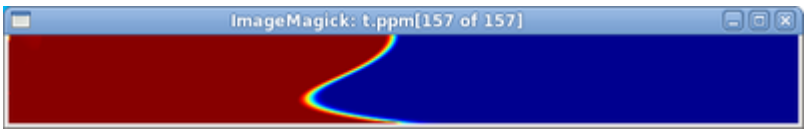
A. Time = 0.



B. High tide (~6.4 hr).



C. Low tide (~12.2 hr).



D. Final low tide (24 hr).

Implementing bottom roughness elements

The previous simulation demonstrates that a no-slip bottom can produce a logarithmic current profile but there is insufficient vorticity to mix the water column well. The result is obviously

sensitive to the viscosity, which was 0.0078 kg/m/s in those simulations ($\mu \sim 0.001$ kg/m/s for water at 20° C). The resulting profile displays a boundary layer that extends over half of the water column. This is not realistic for surface flows in water. For example, the turbulent boundary layer thickness, $\delta = 0.382\ X\ /\ (\mathrm{Re}^{0.2})$. For the problem at hand, Re = (1000)(0.7)(900)/(0.0078125) ~ 10^8, $X = 4 \cdot 900 = 3600$, and thus $\delta = 55$ m. This is ~6% of the flow depth. We note that the refinement for this simulation is $2^6 = 1/64$ and thus the highest resolution is 14 m.

If the boundary layer is laminar, we use $\delta = (4.91)X/(\mathrm{Re})^{1/2} \sim 6$ m. This is unlikely given the large Re. There are several ways to calculate Re but it is not important for our purposes because they all predict a much thinner boundary layer. Since we have already determined that the no-slip bottom predicts the boundary layer to be far too thick (~ half the water depth), we can examine the impact of a bedform on this layer. The next simulation uses a single ellipse to represent morphology:

```
GfsSolid (ellipse (4.0, -0.5, 0.2, 0.05) )
```

This will place an ellipse centered on the bottom of the channel that is 180 m long and 45 m high. This is unrealistic for the desired geometry but it demonstrates the impact of vorticity generation by the bottom. This size feature is slightly lower than the predicted boundary layer thickness. Its impact is strongest within the lower fifth of the water column with s slightly logarithmic current profile higher in the water (Figure 7.3). As expected, the simulation with a bed form and a no-slip bottom grossly over-predicts the bottom boundary layer height.

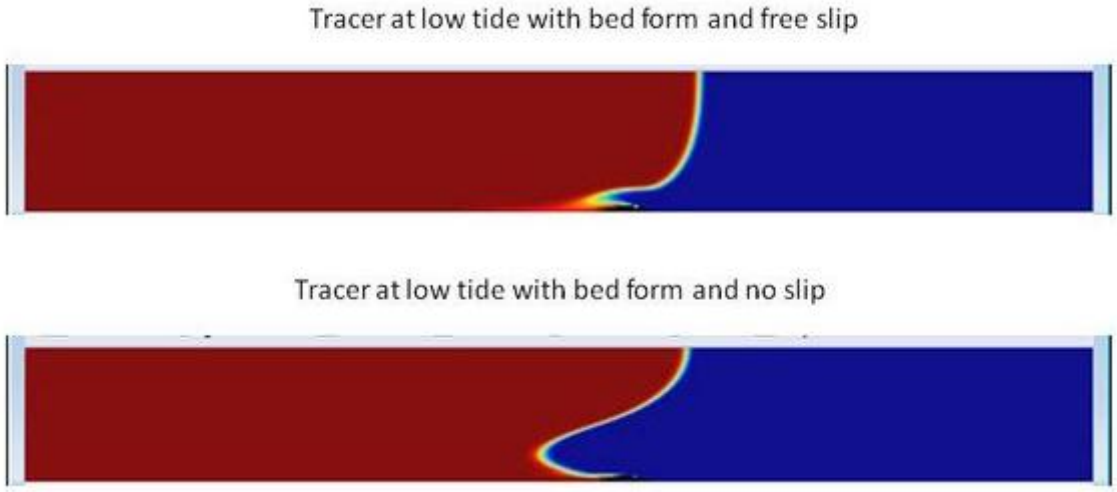


Figure 7.3. Tracer plots at low tide for free slip (top) and no slip (bottom) boundaries.

These results can be scaled for more realistic geometry. For example, we can reduce *h* (water depth) and the viscosity by 10 and achieve the same Re. For this problem, we have a 70 cm/s flow in a 90 m deep channel with a water viscosity = 0.00078 kg/m/s that is appropriate. For this scaling, the bottom feature could be a large tidal sand ridge (18 m long and 4.5 m high).

This would be dimensionally applicable to the Dutch continental shelf (water depth < 100 m), where sand waves are 100 - 800 m long with heights of 1 -12 m and currents exceed 65 cm/s.

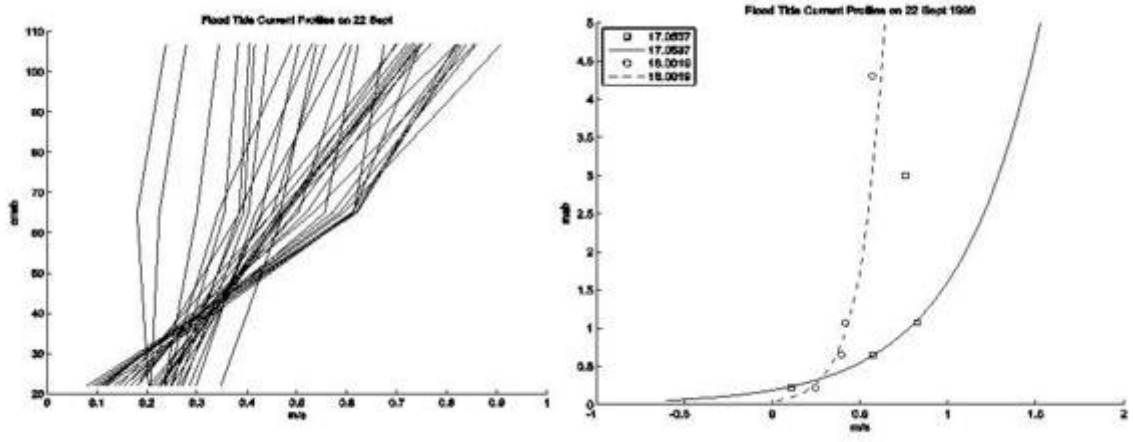
Current profile development

It is unreasonable to simulate the entire tidal cycle in the Tamar River with the CFD model because of the dramatic changes in water depth. This is not a major problem, however, because the data collection periods were limited in duration, lasting less than 6 hr most of the time. The measurements were typically focused on the ebb or flood stages. The water depth during a sampling interval changed by as much as 2.5 m during a spring tide. These changes must be accounted for in simulating hydrodynamics in the estuary, especially for near-surface currents.

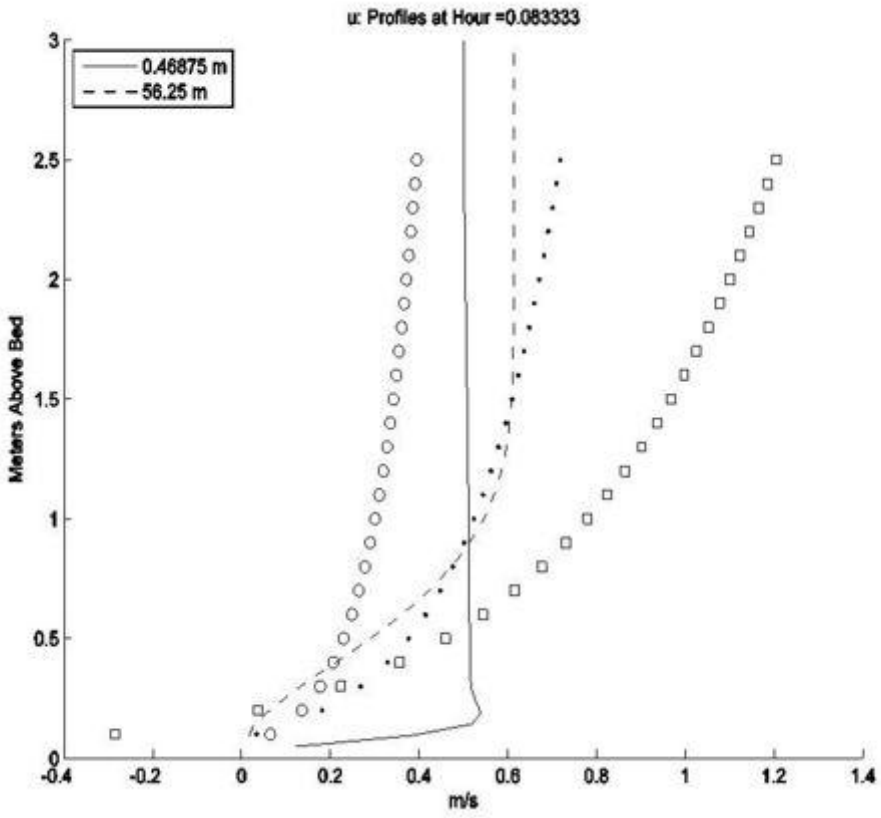
With these cautions in mind, we will examine equilibrium flow conditions using Gerris. These simulations were completed using 25 GfsBoxes and nondimensional input. These simulations incorporated buoyancy forcing as well as an inflow boundary condition. The maximum refinement was 2^6. The size of a box is 3 m, which results in a minimum cell size of 4 cm.

The current profiles measured during a flood tide on 22 September (Figure 7.4A) demonstrate the nature of the flow. This demonstrates the high variability of the flood tide. The right panel shows representative profiles at 17 h and 8 h (squares and circles). It also contains log profile fits using u_* (cm/s)/ y_0 (cm) values of 18.5 / 018.5 and 5.3 / 4.1 for 17 h and 18 h, respectively, when the water depth was 3 m and 4.3 m.

Figure 7.4. Current profiles from the Tamar River, UK.



A. Measured current profiles during a flood tide.



B. Gerris current profile (dash line) and an analytical profile for a rough bottom.

A no-slip bottom and viscosity can be used to create reasonable current profiles in Gerris (Figure 7.4B). The simulation shown here uses a mean inflow current of 50 cm/s and a viscosity of 6.7×10^{-4}. The solid line is near the inflow and the dash line at 56 m along the channel. The roughness parameters, u_* and y_0, are equal to 8.5 cm/s and 8.5 cm, respectively. No roughness elements were used.

Volume of Fluid (VOF) Simulation

The large density contrast between air and water can be simulated using the VOF method in the CFD model. For general ocean circulation in a primitive-equation model, a free-surface is represented instead by the continuity equation with a surface anomaly. Here we will use a density contrast surface as described in the Gerris examples. The justification for the VoF method is seen in height-time plots from the Tamar River (Figure 7.5).

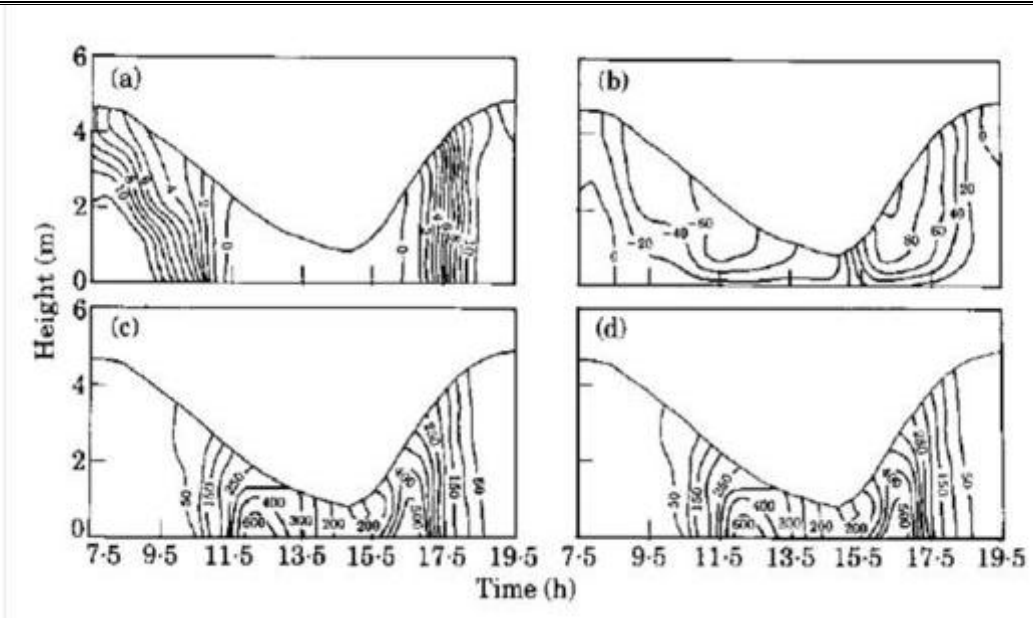


Figure 7.5. Height-time plots of observations during a spring tide on 7 July 1982.

These simulations start using the Rayleigh-Taylor instability and the Cargo vessel wake problems as templates. There is no transformation of units.

The VOF method was examined for an initial sinusoidal surface (Figure 7.6). The domain consists of 4 GfsBoxes (e.g., 4 m). The ratio of upper:lower fluids is 1.2:1000 (air:water). The VOF variable tracer is filtered using *GfsVariableFiltered* because of the high density contrast. The surface has an amplitude of 5 cm and is located at 50 cm above the bottom. Gravity is introduced as a source for V = 9.81. The dynamic viscosity of the fluids is 0.001 and $1{\cdot}1\times10^{-6}$. The ends are closed and the bottom is no-slip, whereas the top is a Neumann BC for *P* and *T*. The result can be interpreted as waves of 5 cm waves with wavelength = 1 m at the ocean surface.

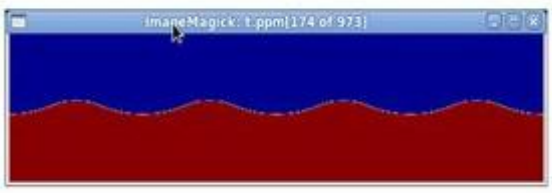


Figure 7.6. Interface between water (red) and air (blue) using the Volume of Fluid method in Gerris.

Simulations with a single GfsBox were completed to examine the dimensionality of the problem. If we want our domain to be 100 m in length but the mean water depth to be only 3 m, we need to initialize the surface to be (1 m + 4 m(tidal range) / 2) /100 m, or y = 0.03. Thus, the initialization for the VOF variable is

```
InitFraction {} T (- (y + 0.03))
```

because *y* is from -0.5 to 0.5. This is the initial still water level. The tidal amplitude is represented by A = 2 m/100 m = 0.02, but we cannot use this as a boundary condition in the VOF method (yet). The open boundary on the left is represented by

```
left = Boundary {
  BcDirichlet U ( UT · sin(2π · t/ TT ))
}
```

where UT and TT are adjusted to get the correct values compared to measured currents and tidal period, respectively. The gravity flux is (hopefully) introduced by (9.8 · 100)/2002. We use 200 because a 0.5 m/s current will cross the 100 m box in 200 s; thus, *G`* = -0.0245. The tidal period becomes (12 · 3600) / 200 = 216 time steps. We will discuss reduced gravity in more detail with respect to the lock-exchange test cases. The result (Figure 7.7) demonstrates the feasibility of the VoF method for the air-water interface. We will attempt to implement this technique using an equation of state for the water (red) as has been discussed in the Lock-exchange tests.

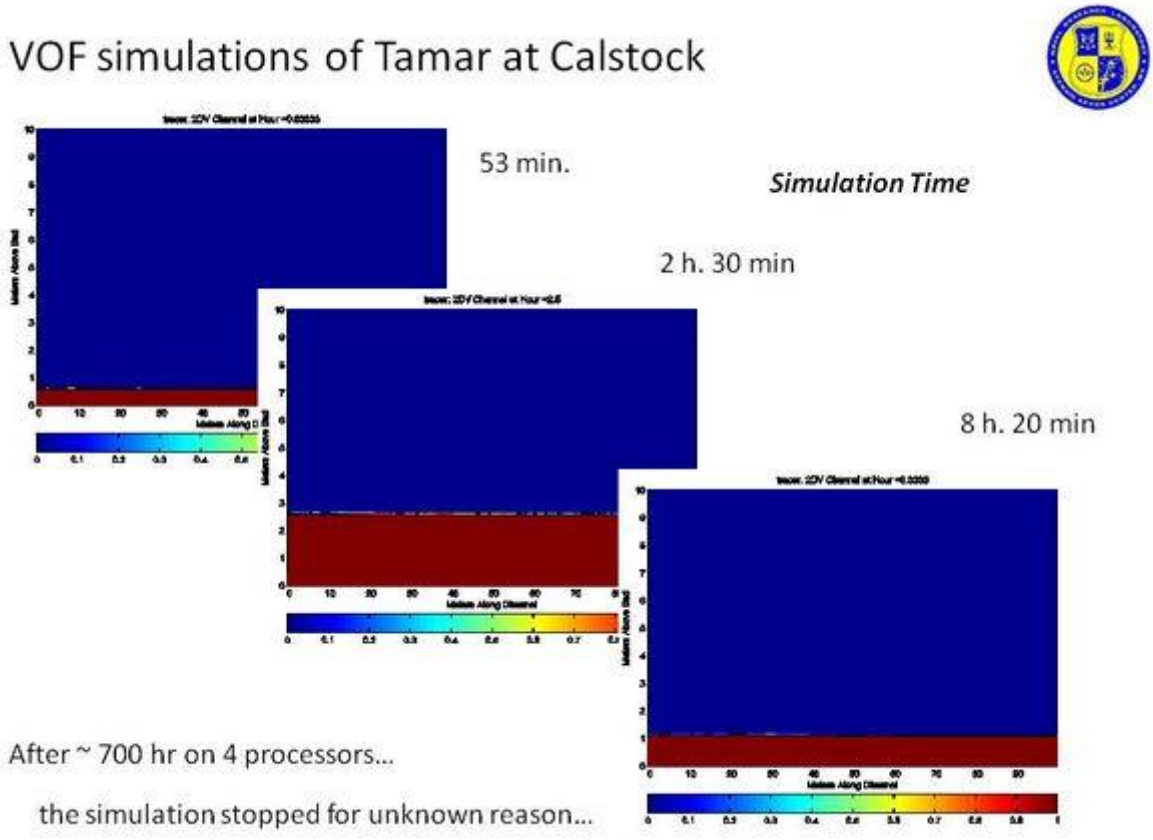


Figure 7.7. Sequence of water depths from VoF simulation for a 12 hr tide.

Section 8: Model Setup

Using the *GfsOcean* Module

Introduction

Gerris can deal with arbitrarily complex solid boundaries embedded in the quad/octree mesh. The geometry of the solid boundaries is described differently for the *Ocean* and *River* modules. The steps for setting up a simulation are somewhat different for the two modules. They will thus be discussed separately. There are three main components to a coastal simulation: (1) domain; (2) tidal boundary condition; and (3) wind forcing. These are discussed in this section. There are examples of both modules in Section 10.

Domain Definition with GTS Files

This is described in detail in Section 4.

Tidal Boundary Condition

Two tidal boundary conditions have been applied in examples: (1) input values from the simulation file, either constant or using an analytical function to describe spatiotemporal variations in tidal amplitude and phase as well as mixed tides; and (2) input of single constituent tides from a standard database using **GTS** files. These methods are explained in Section 5.

Surface Forcing with Wind

The method that is being used so far is to include a wind stress as a uniform, constant source of velocity as described in Section 5.

Using the *GfsRiver* Module

For the GfsRiver module (2D Nonlinear SWE) the bathy surface is defined with a Gerris terrain database (KDT). The basic description of GTS and KDT files is presented in Section 4. The extensive library of KDT database files is discussed as well. This module has been used primarily for tides. This boundary condition is discussed in Section 5. There are examples in Section 10.

Section 9: Lock Exchange Simulations

Introduction

Extensive simulations have also been completed for the lock-exchange problem because it is considered a robust test of a CFD solver. This test has been abused and misused, however, to the point where I felt it was necessary to find out what the authors did to arrive at their published results.

Gravity currents are driven by density differences in a fluid. The resulting pressure gradients are responsible for cold-fronts in the atmosphere and turbidity currents in the ocean. Numerous theoretical and numerical studies have been completed to clarify the inner structure of gravity currents (Hartel et al., 2000), their propagation speed (Maxworthy et al., 2002), and their mixing with ambient fluid (Benjamin, 1968). The laboratory experiments (Figure 9.1) have been limited in the range of conditions that could be reproduced. This has led to the use of numerical models that solve the Navier-Stokes (N-S) equations with different simplifying assumptions (e.g., incompressible and Boussinesq). The goal of much of this modeling work has been to extend the laboratory experiments using direct numerical simulation (DNS) of the N-S equations. This is difficult because the model grid must resolve the smallest dissipative scales up to the integral scale. The smallest scale is the Kolmogorov microscale, which is less than 1 mm for typical flows (O'Callaghan et al., 2010).

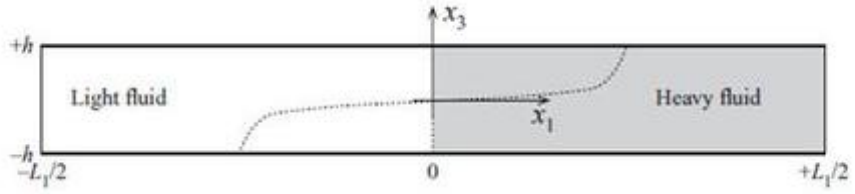


Figure 9.1. First of two physical and numerical lock-exchange domains discussed in this report (Hartel et al. (2000).

The most pertinent applications of Gerris with respect to the Tamar Estuary are the surface plume experiments of O'Callaghan et al. (2010). It seems likely that the section on the lock-exchange test was added to the O'Callaghan paper at the request of a reviewer because it is not well written and has some inconsistencies that will be discussed below. The initial purpose of our tests was to simply reproduce the results from O'Callaghan but this proved difficult because of incomplete simulation reporting in the paper. As I followed up the references within the O'Callaghan paper, I found that they too did not supply sufficient information to unambiguously reproduce their results. Finally, I reached some of the initial laboratory experiments on gravity currents (Simpson and Britter, 1979; Hartel et al., 1997; Maxworthy et al., 2002). I started these tests by reproducing these experimental results as well as I could, because they also did not report all of their test conditions but only summarized them in plots of dimensionless parameters.

Ocean processes are not a typical application of CFD models, which are often used for nondimensional simulations of laboratory or engineering problems. This was one of the major problems in understanding and reproducing published results. The next section is thus intended to introduce some of the relevant parameterizations used to characterize gravity currents before examining the published examples.

Background

The Navier-Stokes equations can be made nondimensional using the Reynolds number, which represents the ratio of momentum and viscous forces; Re = UL /ν, where U = a characteristic velocity, L = a characteristic length, and ν = the kinematic viscosity (Chorin, 1968). The conservation equation for tracers in Gerris also includes the Prandtl number, Pr = ν/D, where D = molecular diffusivity. The diffusion term in the equations is multiplied by 1/RePr. Flow similarity for a given problem is maintained by changing these variables to maintain the values of Re and Pr. However, the determination of U and L is problematic for complex flows in fluids of different densities (Lindgren, 1956).

A gravity current results from the interaction of buoyancy forces and viscosity. The ratio of buoyancy to viscous forces acting on a fluid is approximated by the Grashof number, which is analogous to Re for buoyancy problems:

$$\mathrm{Gr} = [g \beta (T_S - T_0) L_c^3] / \nu^2 \tag{9.1}$$

where: g = gravity acceleration; β = volume expansion coefficient; T_S = surface temperature; T_0 = bulk temperature; and L_c = length. As with other dimensionless numbers, it has been applied to a number of problems, including flat plates, pipes, and bluff bodies. An analogous form of Gr can be used in natural convection mass transfer, in which case, T_S and T_0 are replaced by $C_{a,s}$ and $C_{a,a}$, respectively (concentrations of species a at surface and in ambient medium). The volumetric thermal expansion coefficient β is then given by:

$$\beta = -1/\rho \, (\partial\rho/\partial C_a)_{T,p} \tag{9.2}$$

where: ρ = fluid density; C_a = concentration of species a; T = constant temperature; and p = constant pressure. The value of β is 207×10^{-6} K^{-1} for water at 20° C. The impacts of (2) on Gr are not discussed by Härtel et al. (1997). They represent temperature as a tracer that is initialized with a continuous function across the front, but do not present a constitutive equation for ρ. It is not, therefore, possible to compute u_b because information on the dependence of ρ on T is not given. They state that their simulations are for light and heavy gases; their Figure 7 refers to lock-exchange experiments with Ar ($\rho_1 \sim 1.7$ g/L) and CO_2 ($\rho_2 \sim 2$ g/L). If we assume their simulations are at 0° C and 1 atmosphere, $\rho' = (\rho_1 - \rho_2)/\rho_2 \sim 0.17$. It is not clear how these gasses relate to the initial T distribution because they have different densities at a constant T, which makes an initial condition for T somewhat redundant. It is possible that the dependence of density on T is used as a proxy for these gases. It is further unclear why their simulations used Pr = 2 because Pr = 0.68 and 2.38 for Argon and CO_2, respectively, at 0° C.

The relationship between momentum and the diffusivities of momentum (viscosity) and heat (temperature) is seen in the product, RePr = UL/ν × ν/k = UL/k, which is the ratio of momentum and heat diffusivity. The 1/RePr coefficient in the diffusion term for *T* in Gerris can thus be interpreted as the thermal diffusivity normalized by the characteristic dimension and speed of the overall flow, which makes sense.

The formulation of Gr is often rewritten in fluid dynamics as:

$$\mathrm{Gr} = [\ (u_b\, h)/\upsilon\ ]^2 \tag{9.3}$$

where: $u_b = (g'h)^{1/2}$ = the buoyancy velocity; g' = reduced gravity; and h = the channel half-depth (Hartel et al 1997). The buoyancy velocity u_b is the maximum propagation speed for a sub-critical gravity wave front. It replaces the thermal expansion coefficient β and the concentration of the species of interest C_a in estimating Gr in Equation (3).

The Navier-Stokes equations can be nondimensionalized using Gr for problems that are dominated by buoyancy forces rather than inertia. For example Maxworthy et al. (2002) use √Gr to nondimensionalize the vorticity equation and √(Gr Sc^2) for the density equation, where Sc = the Schmidt number (ν/*D*), which is analogous to Pr. The numerical models that we are examining (Maxworthy et al 2002; O'Callaghan et al 2010) use Sc = 1 and Pr ~ 7, respectively.

The value of Re must be estimated for gravity current problems because the characteristic velocity *U* is unknown *a priori*. We can estimate the frontal Reynolds number, $\mathrm{Re_f} = (u_f \times h_f)/\nu$, where u_f is estimated from the propagation speed of the gravity wave front, which becomes a constant very soon after release (Hartel et al 1997). The characteristic length L_C in Equation (9.1) is typically taken as either the height of the front h_f or the water depth, *H*. It is also estimated from experiments, but it is usually ~0.5 times the channel depth for the problem of interest. Thus it is often represented by *h* as in Equation (9.3).

Values of u_f obtained from the experimental data of Maxworthy et al. (2002, Fig. 5) ranged from a low of 0.025 m/s (Run 11) to a high of 0.12 m/s (Run 5). These experiments used an initial height of dense fluid of 0.05 m. The slower gravity wave that resulted from Run 11 used a dense fluid (ρ_c) with the same density (1.034) as the bottom of the stratified ambient fluid (ρ_b), which was 0.031 greater than the surface fluid (ρ_0). The faster gravity wave in Run 5 resulted from $\rho_c - \rho_b = 0.082$, and $\rho_c - \rho_0 = 0.116$. It is very difficult to estimate dimensionless flow parameters for these experiments because ρ' is not straightforward to compute. An experimental study of saline water flowing into freshwater with ρ' = 0.004 produced typical gravity wave characteristics of $h_f = 0.022$ m and $u_f = 0.027$ m/s (Simpson & Britter 1979).

The buoyancy velocity introduces the relationship between buoyancy and gravity forces, which can be parametrized using the Brunt-Väisälä buoyancy frequency:

$$N = (-g/\rho\_o \quad d\rho/dz)^{1/2} \approx (g'/H)^{1/2} \tag{9.4}$$

This oscillatory frequency (1/s) is exploited by Maxworthy et al. (2002) in describing the relationship between density gradients and gravity wave frontal properties. The values of $g' = g(\rho_c - \rho_a)/\rho_a$ found using density data from Appendix A (Table 9.1) are 0.14 m/s^2 and 0.95 m/s^2,

respectively, for N = 0.97 1/s and 2.5 1/s. They use an alternative formulation N_C because of their experimental setup (Figure 9.2), however. The values of g' = (ρ_c - ρ_0)/ ρ_0 used in their analysis are somewhat larger—0.303 m/s^2 and 1.13 m/s^2, respectively, and N_C = 1.42 1/s and 2.75 1/s. This buoyancy frequency best describes the current both within and above the gravity flow itself, as the waves generated by the flow reinforce the front; however, the propagation speed of the front is somewhat less than this velocity. Following the physical mechanism by which the front propagates, the buoyancy velocity can also be defined as $u_b = NH$. The value of Pr for water at 20° C is 7.

Table 9.1. Experimental conditions from Maxworthy et al. (2002).

Expt	h/H	$AR = L/h$	ρ_b	ρ_C	ρ_0	N	N_C	R	Fr	NT_{tr}	X_{tr}/h
1	2/3	2	1.032	1.035	1.004	1.351	1.421	1.107	0.255	24	9.18
2	2/3	2	1.033	1.045	1.004	1.374	1.634	1.414	0.375	33.7	18.96
3	2/3	2	1.035	1.090	1.005	1.397	2.352	2.833	0.637	18.7	17.87
4	1/3	4	1.044	1.070	1.005	1.593	2.057	1.667	0.317	16.1	15.31
5	1/3	4	1.037	1.119	1.003	1.489	2.750	3.412	0.565	9.2	15.59
6	1/3	4	1.034	1.075	1.004	1.398	2.151	2.367	0.438	12.4	16.29
7	1/3	4	1.034	1.065	1.004	1.398	1.993	2.033	0.375	12.9	14.51
8	1/3	4	1.033	1.037	1.005	1.350	1.443	1.143	0.182	12.7	6.93
9	1/3	4	1.033	1.049	1.005	1.350	1.702	1.589	0.290	18	15.66
10	1/3	4	1.036	1.045	1.003	1.467	1.655	1.273	0.232	17.6	12.25
11	1/3	4	1.034	1.034	1.003	1.422	1.422	1.000	0.131	9.9	3.89
12	1/3	4	1.035	1.048	1.004	1.421	1.703	1.435	0.269	17.1	13.80
13	2/3	4	1.064	1.099	1.008	1.907	2.433	1.629	0.437	16.6	10.88
14	2/3	2	1.065	1.139	1.008	1.932	2.917	2.280	0.555	15.5	12.90
15	1/3	4	1.065	1.094	1.006	1.958	2.390	1.490	0.287	19	16.36
16	1/3	4	1.064	1.072	1.006	1.942	2.080	1.147	0.190	15.5	8.84
17	1/3	4	1.067	1.112	1.005	2.001	2.638	1.738	0.327	19.6	19.23
18	1/3	4	1.066	1.163	1.008	1.945	3.170	2.655	0.463	13.3	18.47
19	1/3	4	1.065	1.088	1.007	1.941	2.294	1.397	0.264	19.6	15.52
20	1/3	4	1.065	1.122	1.006	1.958	2.746	1.966	0.379	15.3	17.40
21	1/3	4	1.067	1.082	1.006	2.000	2.235	1.249	0.233	10.4	7.27
22	1/3	4	1.067	1.079	1.006	2.000	2.181	1.189	0.210	14.2	8.95
23	1/3	8	1.068	1.085	1.008	1.965	2.232	1.291	0.249	29	21.66
24	1/2	8/3	1.067	1.069	1.008	1.948	1.983	1.036	0.211	18.3	7.72
25	1/2	8/3	1.066	1.082	1.008	1.940	2.202	1.288	0.295	28.9	17.05
26	1/2	8/3	1.067	1.115	1.008	1.957	2.639	1.819	0.415	20.6	17.10
27	1/2	8/3	1.069	1.181	1.007	2.007	3.363	2.808	0.588	20.1	23.64
28	1/2	8/3	1.035	1.042	1.004	1.410	1.582	1.259	0.291	28.2	16.41
29	1/2	8/3	1.035	1.040	1.003	1.444	1.555	1.159	0.267	28.1	15.01
30	1/2	16/3	1.036	1.054	1.003	1.456	1.827	1.575	0.382	18.3	13.98
31	1/2	8/3	1.068	1.139	1.007	1.990	2.927	2.162	0.483	15.3	14.78
32	1	4/3	1.068	1.072	1.009	1.947	2.019	1.075	0.295	60	17.70
33	1	8/3	1.068	1.104	1.007	1.990	2.505	1.584	0.448	29	12.99
34	1	4/3	1.034	1.072	1.004	1.386	2.103	2.302	0.597	21.5	12.84
35	1	4/3	1.034	1.094	1.004	1.398	2.421	3.000	0.701		
36	2/3	2	1.067	1.067	1.0075	1.965	1.965	1.000	0.230		

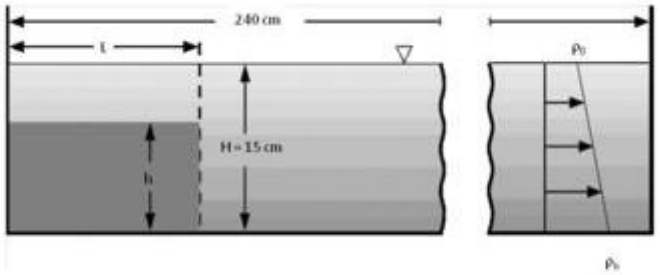


Figure 9.2. Sketch of experimental conditions of Maxworthy et al. (2002)

The ratio $Fr_f = u_f / u_b$ is the frontal Froude number, which represents the observed propagation speed u_f relative to the theoretical limit. In the experiments of Simpson and Britter (1979), Fr_f exceeded 1 for $h_f < 0.2H$; however, their data were restricted to $h_f < 0.3H$. The critical value for Fr_f is $1/\pi$ (0.318) (Maxworthy et al 2002). If the front propagates at this speed, it is in step with the gravity wave that drives it. The theoretical limit for Fr_f is $\sqrt{2}$ for a free-slip boundary (Benjamin 1968). We can calculate Fr_f for Runs 5 and 11 from above to check for consistency. Using N_C for the buoyancy frequency, u_b = 0.21 m/s and 0.41 m/s for the slow and fast gravity waves, respectively. The values of Fr_f for these experiments can thus be estimated as 0.117 (Run 11) and 0.29 (Run 5), which suggests that neither of these gravity flows was supercritical. However, Maxworthy et al. (2002) report a much larger value of Fr_f for Run 5 (0.565) whereas they have a similar value for the slower wave (Fr_f = 0.131). This discrepancy occurs because they used experimental data from a previous study (Rottman & Simpson 1983) to compute Fr_f rather than directly estimating u_b as we have done. Since their scaling analysis was based on experimental results that are substantially different than we are examining, we will use the more direct method discussed by O'Callaghan et al. (2010).

There is a fundamental relationship between Re and Gr that is discussed by previous authors. Several quantitative comparisons are made as well. Despite these attempts to be accurate in applying these dimensionless numbers to the experimental and simulation results, however, there remain some discrepancies that must be explained to interpret their results correctly. A comparison of the momentum equation in its velocity-pressure formulation solved in Gerris (O'Callaghan et al 2010) and the vorticity-stream function form applied by Härtel et al. (2000) shows that Gr is equivalent computationally to Re^2. The motivation for using Gr instead of Re is its dependence on g' and thus the density distribution, rather than measured values of u_f and h_f. This difference is explicitly used by Härtel et al. (1997) in characterizing the lock-exchange problem. They restrict their discussion to Re_f and Gr as defined in Equation (9.3).

Simulations with a Non-Hydrostatic 2D Model

Preliminary lock-exchange simulations have been completed with a nonhydrostatic 2D (NH2D) model:

- 2nd-order Adams-Bashforth in time.
- 2nd-order centered advection with Laplacian diffusion.

- Diffusion: background value of 0.01 cm2/s,
- same in both x and z. Background value is increased
- to maintain a max grid-cell Re Number = 10, i.e.,
- $K = \max[K_0, u \cdot dx/10]$

The results from simulation *LEX#10* can be viewed in this animation file: **lex10.fli**. The following parameters were used.

- 2DV domain is 2 × 0.3 m
- inflow on the left
- dx = 1 mm
- dt = 0.001 s
- number of cells = 2004 × 301
- $K_m = 10^{-6}$ m^2/s (0.01 cm^2/s)
- $K_S = 10^{-6}$ m^2/s (0.01 cm^2/s)
- Max cell Re = 10
- $\beta_S = 7.418 \times 10^{-4}$ (volume expansion for salt)
- S fields saved every 10 iterations (0.01s) for 1 m (half-length)

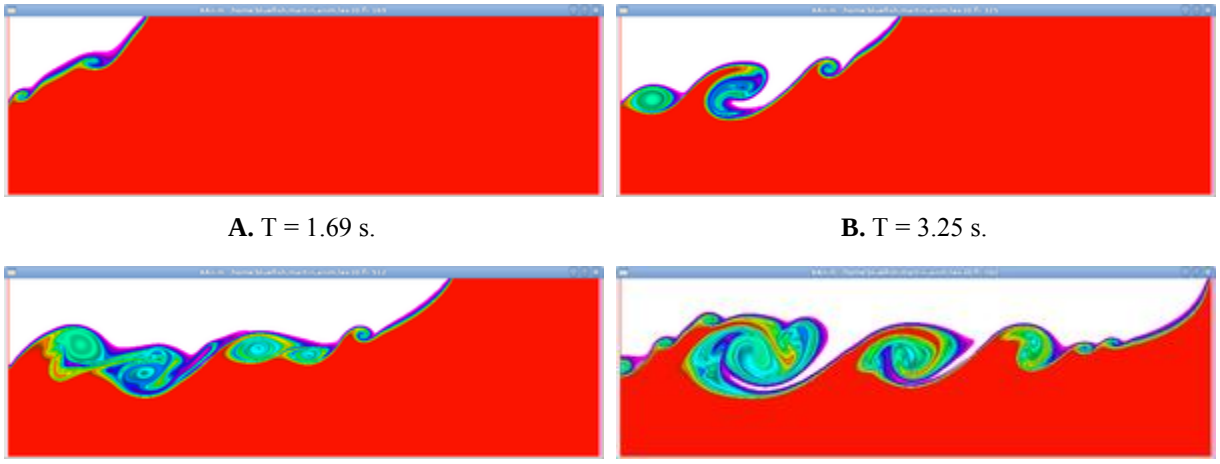
Initial Condition:

- constant T
- $\Delta S = 40.44$ psu
- $x_F = 4$ mm (half-width of front)
- $S(i,k) = 0.5 \cdot \Delta_S * \tanh[(i-xic)/4.0]$
- xic = 2005/2

Results

With the high grid resolution, low viscosity/diffusivity, and thin initial frontal width, the Kelvin-Helmholtz (K-H) rolls are well formed at ~2 s (Figure 9.3A), and the salinity patterns become quite elaborate at later times. Some of the details of the smaller-scale flow patterns are "hidden" in the larger scale flow. It is necessary to plot the stream function in a frame of reference moving with the smaller-scale feature in order to see the circulation associated with the feature. Note that the larger-scale flow may have a vertical as well as horizontal component, so that shifting the point of reference in just the horizontal may not be sufficient to see the local circulation associated with the smaller feature.

Figure 9.3. Salinity distribution for run LEX#10.



A. T = 1.69 s. **B.** T = 3.25 s.

C. T = 5.12 s. **D.** T = 7.02 s.

The initial salinity range is +/- 20.22. Before 5.88 s (Figures 9.3B and 9.3C), the salinity remains in the range of +/- 21.0, i.e., the advective overshoots are small. At later times (Figure 9.3D), however, the range increases to +/- 24.0 and sometimes more. There may be problems with the bc at the end walls, or just trouble with advective overshoot in the corner when the front hits the corner. At 10s, the adv CFLs are 0.266 in *x* and 0.172 in *y*;, i.e., it seems the timestep could be doubled, although I know that the AB2 advection does not like having a CFL over 0.5. At earlier times, the vert adv CLF sometimes exceeds 0.4.

Discussion

Look at the visous/diffusive limit for water:

- kinematic viscosity = 0.010 cm^2/s data from Bachelor, p 597
- thermal diffusivity = 0.00142 cm^2/s values are for T = 20 C
- diffusivity salt = 0.00014 cm^2/s

Requirement for viscous limit: Cell Re = 10 = u·dx/ν. With max u = 20 cm/s, and ν = 0.01 cm^2/s, we need dx = 0.005 cm. For salt we need 70 times higher resolution, i.e., ~ 0.0001 cm. For this experiment, minimum mixing coefficients are 0.01 cm^2/s, and max mixing coefficients are ~ 0.2 cm^2/s.

The question of what happens in reality (i.e., if we could perform the actual experiment), depends on the initial frontal thickness and any initial perturbations that exist. It might be impossible to perform the actual experiment to look at the initial instabilities, i.e., what happens initially would depend on things you could not control sufficiently well.

Note that a (vertical) length scale is set by the critical Richardson Number (R_c). Given the salinity and velocity differences across the interface, R_c will define a length scale

- $R_c = g\cdot\Delta\rho\cdot L_z/(\rho\cdot(\Delta v)^2) \rightarrow L_z = \rho\cdot(\Delta v)^2 \times R_c/(g\cdot\Delta\rho)$

During the initial acceleration of the fluid, the small velocity differential across the interface will generate a small vertical scale and small K-H rolls. The final velocity, which is related to the frontal propagation speed (which depends ~ on the internal wave, IW, speed), will define the eventual thickness of the interface region for a propagating plume in the region behind the front. In this case, with $\Delta\rho = 0.03$ gm/cm^3, a final velocity difference across the interface of $2\cdot 16 = 32$ cm/s, and $R_c = 0.25$, the interface thickness will be ~8.7 cm.

Defining the interface thickness in terms of R_c, however, begs the question of what is actually going on, in that R_c is a result of the action of the K-H instabilities. There is still the more basic question of understanding the K-H instability itself. In the literature it is said that the shear-instability becomes turbulent for about Re > 300. For this experiment, $Re \sim u\cdot L_z/K = 20*10/0.2 = 1000$. Hence, the results agree with the theory that the K-H rolls will develop into turbulence for Re = 1000. If I were to increase the viscosity to 1 cm^2/s so that Re = 200, we should expect that the K-H rolls would NOT become turbulent.

Since the velocity depends on the internal wave (IW) speed, which depends on the stratification and depth, the length scale for this problem, as defined by the Richardson Number, can be expressed in terms of just the density stratification and the depth of the channel. For a channel of depth, *H*, with fluids of different density in layers of thickness $H = H_1 + H_2$:

- $L_z = \rho\cdot(\Delta v)^2 * R_c/(g\cdot\Delta\rho) \quad R_c \sim 0.25$
- IW speed for 2-layer system:
- $c^2 = g\cdot(\Delta\rho/\rho)\cdot(H_1\cdot H_2)/(H_1 + H_2)$

For $H_1 = H_2 = H/2$:

- $c^2 = g\cdot(\Delta\rho/\rho)\cdot 0.25\cdot H$
- $(\Delta v)^2 = (2\cdot c)^2 = g\cdot(\Delta\rho/\rho)\cdot H$

Therefore:

- $L_z = H\cdot R_c$

Hence, we get the curious result that the vertical scale of the interface thickness for stability is independent of the density gradient and is proportional to the depth. Note that this is consistent with all my results for this problem, i.e., the vertical scale of the largest K-H rolls, relative to the channel depth, is fairly constant.

For the case of a thin plume near the surface ($H_1 << H_2$), where the lower layer velocity is small, we get:

- $c^2 = g\cdot(\Delta\rho/\rho)\cdot H_1$
- $(\Delta v)^2 = c^2 = g\cdot(\Delta\rho/\rho)\cdot H_1$

Therefore:

- $L_z = H_1 \cdot R_c$

This scaling of the K-H mixing is consistent with my results for a thin plume (SPF #43-47). (Note - the thin plume expts needed higher resolution.) An unresolved question remains, however; what about the timescale of the evolution of the K-H instability?

Simulations with Gerris (2D CFD)

The preliminary experiments discussed above lead directly to a more formal series of simulations designed to examine the suitability of this idea to reproducing buoyancy flow in an estuary. These are fully discussed on the Gerris Lock Exchange page.

Before proceeding, however, it is instructive to jump ahead slightly and apply some of the concepts that will be applied in these experiments to the experiment with section 9.3. First, we can apply the concept of the Grashof number, which is analogous to the Re number for buoyancy flows:

$$Gr = ((u_b \cdot h)/\nu)^2$$

where the buoyancy speed u_b is analogous to the propagation velocity c for internal waves. We can thus estimate u_b from the supplied conditions:

$$u_b = [g \cdot \Delta\rho/\rho \cdot h]^{1/2}$$
$$= [(9.81 \text{ m/s}^2) \cdot (0.03) \cdot (0.15 \text{ m})]^{1/2}$$
$$= 0.21 \text{ m/s}$$

where the thickness of the gravity wave is estimated as $h = H/2$ (0.15 m) following Hartel et al. (1997) or 0.3 m. We have used the former in this case for consistency with the Gerris results. We can estimate the gravity wave front propagation speed u_f from the salinity distribution predicted by the NH2D model. The internal wave front reached the end of the channel in ~7 s, which indicates u_f = (1 m)/(7 s) ~ 14.5 cm/s. We used this value because it is very likely that the length of the domain refers to one-half of the channel length (see Hartel et al., 1997).

The Grashof number is estimated from:

$$Gr = [(0.21) \cdot (0.15) / 10^{-6}]^2 \quad \sim 10^9$$

and the Froude number of the front:

$$Fr_f = u_f / u_b$$
$$= (0.145)/(0.21)$$
$$= 0.7$$

This is slightly larger than other model results for a free-slip bottom but it is lower than the theoretical maximum for naturally forced buoyancy. Buoyancy driven flows cannot exceed Fr = $1/\sqrt{2}$ ~ 0.707 (Hartel et al., 2000). Results from Gerris (no-slip bottom) for this approximate value of Gr indicate that Fr is about 0.65.

More complex computations are available but they give the same answer. A calculation with program *igw_modes* shows the IW speed for a two-layer flow with an interface thickness of 2 cm, $\Delta\rho = 0.03$ g/cm^3 and depth = 30 cm, is about 14.4 cm/s. Hence, the wave should propagate to the edge of the domain in just over 7 sec. With an interface thickness of 10 cm, the phase speed of the IW decreases to 13.0 cm/s.

This brief comparison demonstrates that the behavior of an internal wave/buoyancy front is equally described by different parameterizations of the simulation and flow. The expected uncertainties in the exact values of these properties are due to the nature of these perturbations.

Hartel et al. (1997)

We begin our simulations of previous lock-exchange reports with the earliest that appears to be directly relevant to the present work. We would like to use Simpson and Britter (1979) but their experimental apparatus is not readily simulated and hasn't been reproduced in previous model studies either. The experiments of Hartel et al. (1997) will be reproduced as completely as possible. As suggested by the previous discussion, however, we shall be satisfied with similar-looking flows to those from research papers. This does not mean that the original works lacked detailed analysis but only that the published reports were not intended as standards.

The model results of Härtel et al. (1997; hereinafter H97) (Figure 9.4) can be used to check our results for consistency. We note that the results in the figure are probably for gases, Ar and CO_2. This should not be a problem if the dimensionless approach is valid. We must keep this in mind when evaluating these results. In order to confirm our speculation that the use of *T* in their study was a proxy for solving the DNS problem for two gases, we have completed a series of experiments with Gerris to attempt to reproduce their results (Figure 9.4).

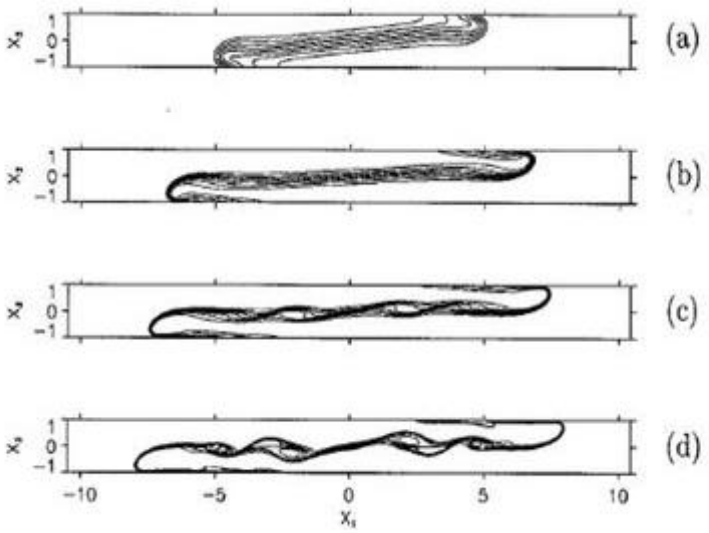


Figure 9.4. Results for (a) Gr = 2.5×10^3 (b); 2.5×10^4; (c) 10^5; (d) 6.125×10^5 (Härtel et al., 1997).

For these simulations, h = 0.1 m, the length of a GfsBox, L = 0.2 m, there are 12 GfsBoxes, and GfsRefine = 6 (i.e., minimum dx = 0.2×2^6 = 3.1 mm). We will calculate Gr = $((u\_b\times h)/\nu)^2$ using different combinations of parameters. Note that our simulations have the denser fluid on the left rather than the right as in the original work. The experimental parameters include the dynamic viscosity *mu* and the molecular diffusivity of heat (mass), D. These are listed because they are the parameters included in a Gerris simulation file. The kinematic viscosity $\nu = mu/\rho$. As discussed above, we cannot compute the exact flow parameters from the information given in H97. Furthermore, we are using dimensions based on the discussion. We include diffusivity for our tracer (salt) that is 1/7th the value of ν (i.e, Pr = 7).

Our results (Figure 9.5) are presented for approximately the same values of Gr as in Figure 6 of H97. Equation (9.3) is used to calculate Gr to compare to the original work. These simulations use ρ' = 0.001 and adjust Gr using the viscosity ν. We used Pr = 7 because we are primarily interested in water. This is much less mixing than Pr = 2 as used by H97.

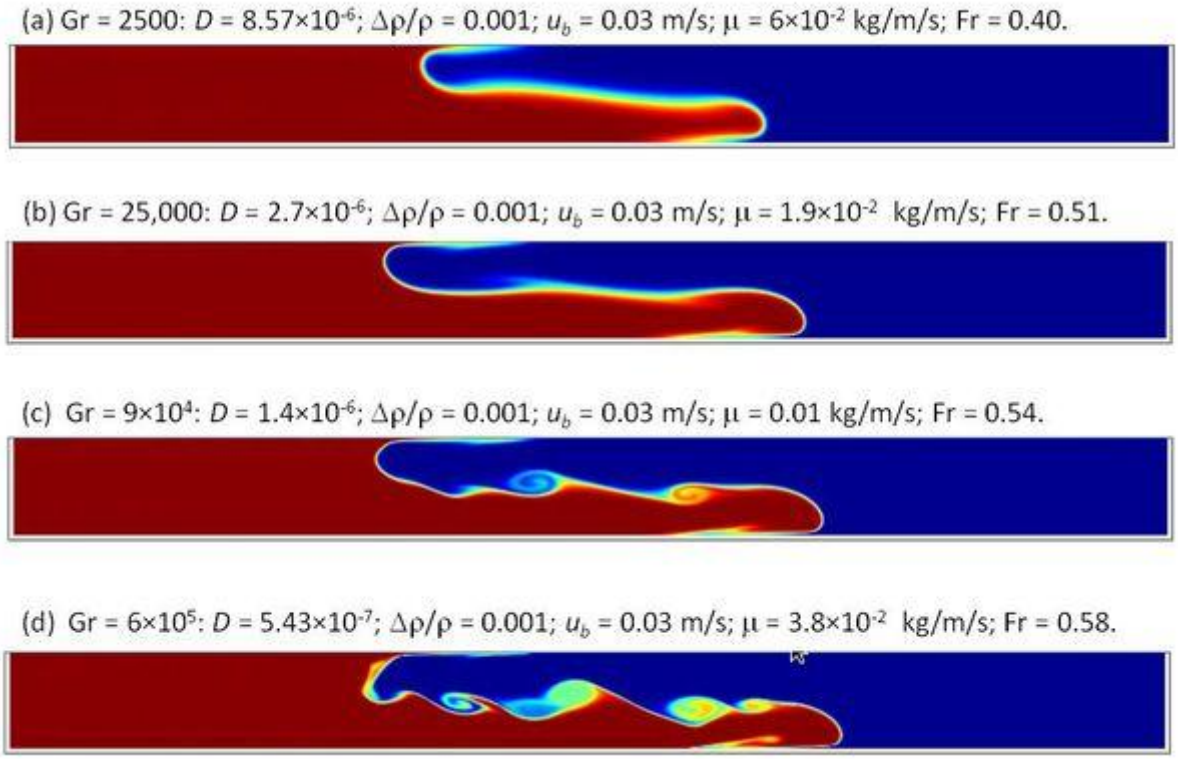


Figure 9.5. Summary of NRL results for H97 at t = 30 s (see Figure 11.4).

The first experiment (9.5A) has Gr = 2500 by setting ν = 0.00006 m^2/s. A reasonable mixing band is attained along the interface between the fluids with $D = 2\times10^{-3}$ m^2/s (Pr = 0.5). This compares well to H97 (Figure 9.4A). The tracer section at t = 30 s shows somewhat less mixing than H97 because the molecular diffusivity D was not reported. This diffusion can decrease u_b for small Gr. This mixing is partly a function of Sc = ν/D, which unlike Pr, is not an inherent material property; it depends on the chemical species, temperature, salinity, and pressure. The

values of D (molecular diffusivity) for Na^+ and Cl^- (separately) at 20° C and 10 PSU are ~0.11 and 0.12 m^2/s, respectively (Boudreau 1997). With the values of diffusion and kinematic viscosity (μ /1000) I used, Sc (~ μ /(D×10^3) varied between 0.5 and 20 to get reasonable results.

The value of Gr increases to 25,000 when ν is reduced to 0.000019 m^2/s (Figure 9.5B) and a result similar to that from H97 is produced (Figure 9.4B) but with less diffusion. Note that Fr_f has increased from 0.4 to 0.51, which is still too low for Figure 9.4 (Hartel et al., 2000). H97 completed numerical simulations for Gr = 10^5 (Figure 9.4C). Gerris predicts Kelvin-Helmholtz instabilities that are a little better developed because of our decreased mixing. Our result for Gr = 6.125×10^5 (Figure 9.5D) is very similar to H97 (Figure 9.4D) with the better-developed turbulence as discussed above.

We can further evaluate the simulations numerically by comparing the value of $\Delta x = 2^6 = 0.0031$ m to the recommended value of $(Gr\times Pr^2)^{-1/4}$ for a DNS (H97), for which we get 0.0029 m. This value is close enough that the monotonically integrated large eddy simulation (MILES) approximation is more than adequate (Popinet et al., 2004). We note that the Kolmogorov scale ($\sim 1/Re_f$) ~ 0.000625 m, which we can use to estimate a dimensional value using L = 0.2 m, or 0.12 mm.

Hartel et al. (2000)

The next series of simulations are from another modeling study of the original laboratory experiments (Hartel et al 1997; Hartel et al 2000). These results (Figure 9.6) analyze the behavior of the gravity-current head and flow topology for Gr = 1.25×10^6, 1.5×10^6, 4×10^8, and 2×10^9 in a symmetrical lock domain as shown above. This extends the results of Hartel et al. (1997) to stronger buoyant forcing. The original Boussinesq numerical model is used by the authors in addition to a Fourier solution of the same equations. I have limited our Gerris simulations to realistic fluids.

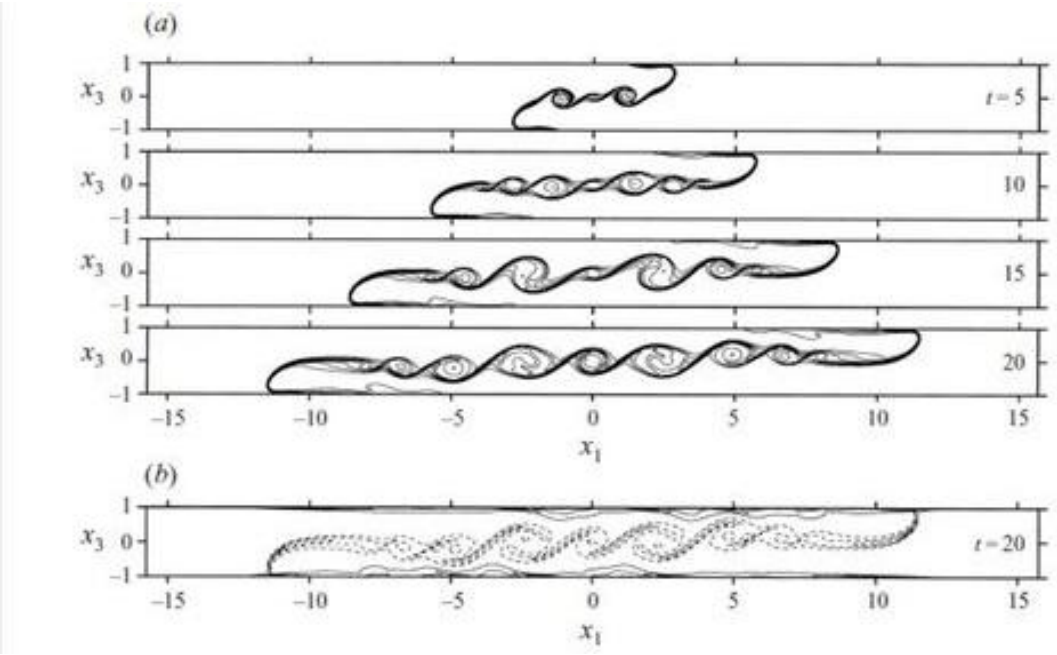


Figure 9.6. Development of 2D flow at Gr = 1.25×10^6 (Hartel et al., 2000).

The first flow examined is for Gr = 1.25×10^6 (Figure 9.7). Since the objective of our work is to better understand gravity flows in brackish water, we would like to use reasonable dimensional values of ρ', viscosity, and diffusivity. Our simulations were for Gr = 1.33×10^6 with *mu* (dynamic viscosity) = ~0.0026 kg/m/s and g' = 0.00981 m^2/s. For the first simulation we used Pr (Sc) = 7. The gravity flow contains the same number of vortices as H00 (Figure 3), but they are less regular because of the reduced diffusion. Positive vorticity (yellow) is generated along the fluid interface and negative vorticity (blue) is created at the solid boundaries. H00 used much more diffusion as represented by Pr (Sc) = 0.7. Our simulations with this ratio were very similar but had the expected wider interface between fluids.

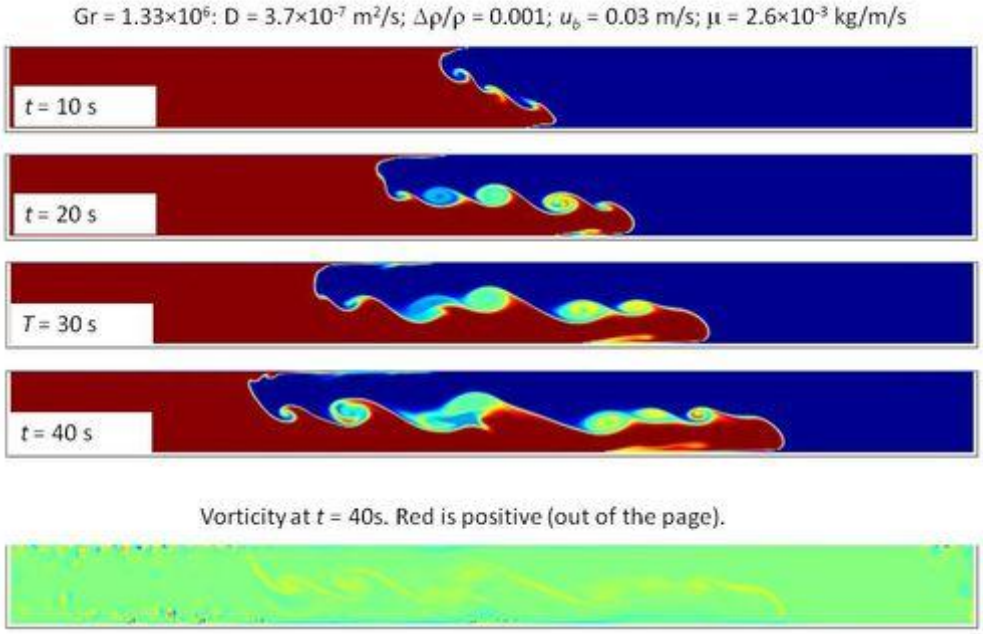


Figure 9.7. Gerris simulation results for evolving gravity flow at Gr = 1.33×10^6 for Pr (Sc) = 7.

The head of the gravity flow is of interest because of its impact on the entrainment of ambient fluid into the flow. H00 examined the gravity flow head structure for Gr = 4×10^8 and 2×10^9. The fluid properties from H00 are not given. The results shown here (Figure 9.8) demonstrate the smooth, elongate nose at Gr = 4×10^8 in both models. It was very difficult to exactly match the higher Gr from H00 because the result is very sensitive, so the results below are for Gr = 7.7×10^9; however, they are sufficiently similar to demonstrate that the model is responding to the large density gradient correctly. Hartel et al. (2000) computed u_f by estimating the position of the front but they do not report these data. They go on to plot Fr_f as a function of Gr (Figure 4 in H00). They define Fr_f as the ratio of the asymptotic front velocity, u_f, to the buoyancy velocity u_b, which is estimated by Equation (9.4).

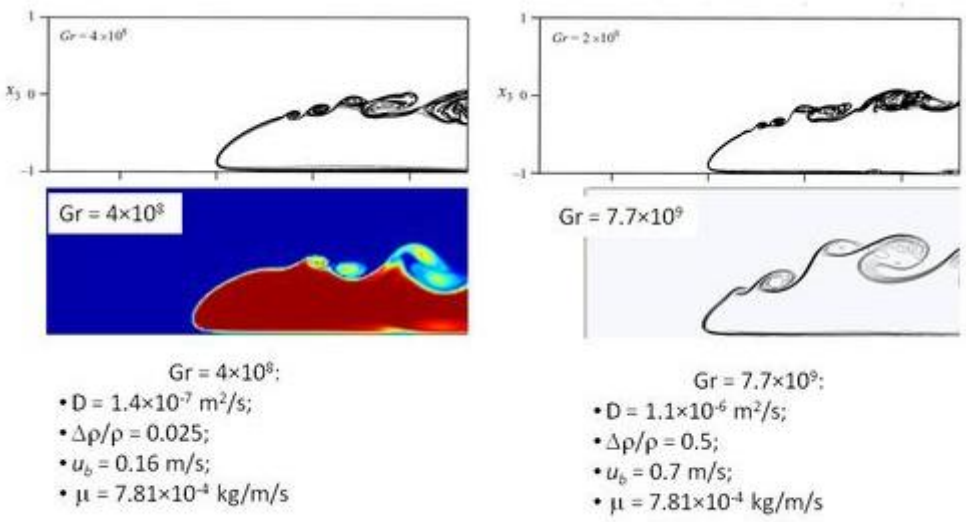


Figure 9.8. Comparison of H00 (top) and NRL (bottom) results at similar Gr numbers.

Our results (Table 9.2) fall along the no-slip line from H00 (Figure 9.9) but we have extended the plot slightly with the following simulations.

Table 9.2. Summary of NRL Experiments.

Gr	**Fr_f**
2.5×10^{3}	0.40
2.5×10^{4}	0.51
9.0×10^{4}	0.54
6.2×10^{5}	0.58
2.9×10^{7}	0.65
1.6×10^{10}	0.66

The value of $Fr_f = 0.66$ for the last case in the table would fall nicely on the extrapolated solid line from Figure 9.9.

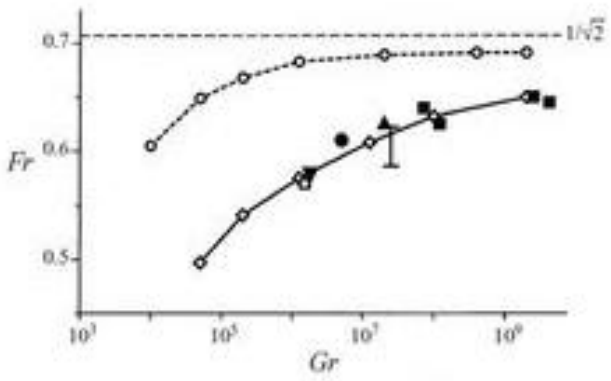


Figure 9.9. Plot of results for Fr as a function of Gr from several sources (Hartel et al., 2000).

Maxworthy et al. (2002)

This section began by examining the experimental results of Maxworthy et al. (2002) (hereinafter M02), but it was not possible to fully reproduce their numerical simulations because of peculiarities of their apparatus (Figure 9.2). They used a stratified ambient fluid with a denser fluid released from an area that was not necessarily the full height of the tank. The density ratio is more complex to calculate for this setup; $R = (\rho_c - \rho_0)/(\rho_b - \rho_0) = N_C^2/N^2$, where ρ_c = density of heavier fluid; ρ_0 = density of ambient fluid at surface; and ρ_b = density of ambient fluid at bottom. Here $N^2 = (g/\rho_0)(-d\rho/dz) = g(\rho_b - \rho_0)/ \rho_0 H$. Furthermore, $N_C^2 = g(\rho_c - \rho_0)/ \rho_0 H$ (Equation 9.4). The difference between these two versions of the buoyancy frequency is the use of the bottom ambient density for N versus the density of the fluid in the lock for N_C. Thus, R is the ratio of the density differences for the lock and ambient fluids. For a large contrast in these density differences (i.e., R large), the density difference can be given by, $\rho_c - (\rho_b + \rho_0)/2$. For this case, the average density of the ambient fluid is used. R needs to be substantially larger than this average to create a gravity flow. This is an issue because the only visualization of a flow simulation (Figure 9.10) is for an unlabeled run with parameters not listed in their Table 1.

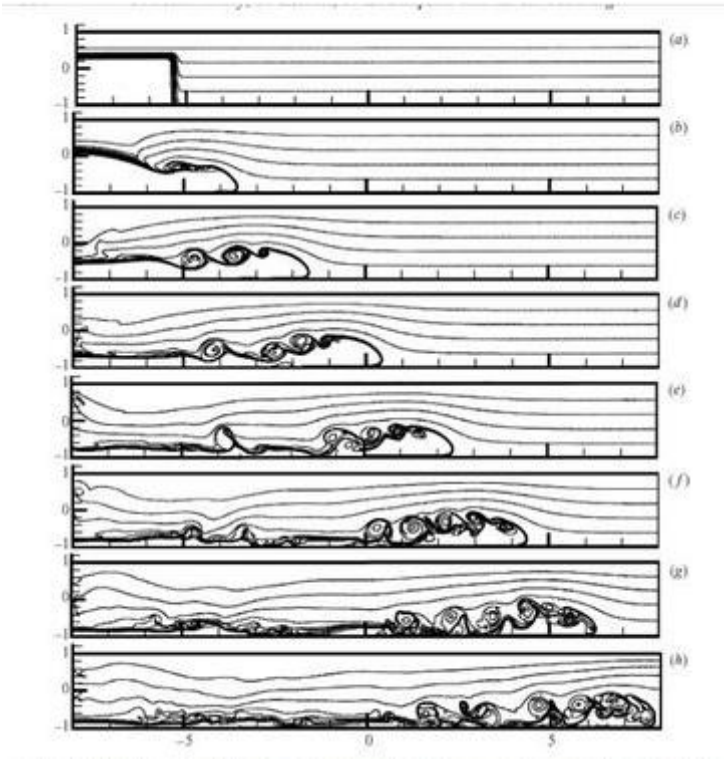


Figure 9.10. Results from Maxworthy et al. (2002), showing gravity flow evolution from a laboratory experiment.

This experiment was reproduced using Gerris with estimates for the appropriate fluid properties. The simulation uses dimension,L_c = 15 cm and there are 16 boxes. This gives a total length of 240 cm. This box size matches the length of original laboratory tank but it is only half as deep because of an inconsistency in the description of the experimental apparatus in M02. They label the fluid depth as H = 15 cm in the caption for their Figure 1. In the text, the say the tank is 30 cm deep. Their figure 9 (reproduced here as Figure 10.10) says the numerical calculations are scaled with $H/2$ and H = 15 cm. What the figure caption should have said is, $H/2$ = 15 cm. We can verify this using the relationship between $\mathrm{Fr_f}$ and u_f: $u_\mathrm{f} = \mathrm{Fr_f} \cdot u_b \sim \mathrm{Fr_f} \cdot N{\cdot}H$ = (0.489)(1.981)(0.15) = 0.15 m/s. This speed can be estimated from Figure 9 (M02) to be 31 cm/s, which is twice the estimate. This discrepancy cannot be reconciled with the available data. The photographs of laboratory experiments are unlabeled. Figure 10.10 may not show the entire model domain. If we assume that H = 15 cm, the value of u_f is half and matches the estimate.

With the uncertainty associated with the M02 report, I first reproduced the results for the slower speed using a reduced value of gravity in the simulation; these results (Figure 9.11) are in good agreement with the *slower* interpretation of M02. The measured u_f is 14.2 cm/s, which is very close to that from M02. A large density contrast was required to match their simulation. This gravity-flow head velocity is similar to that from the previous experiments from Härtel et al. (1997; 2000). These are not directly comparable, however, but they are similar enough to

demonstrate that unrealistic ocean density gradients are required to reproduce many of these results. From their paper, we do know that Maxworthy et al. (2002) used water but they did use unusual compounds to adjust the density as much as they did. For our purposes, we have succeeded in reproducing their experimental results. We are not going to try and reproduce everything else, however. A full value of gravity was also used and had the expected result of approximately doubling the value of u_f to 27 cm/s. These results are not shown.

ρ_c	ρ_0	ρ_b	h/H	u_f(cm·s^{-1})	μ	D
1.61	1.003	1.067	2/3	14.2	0.00078125	10^{-6}

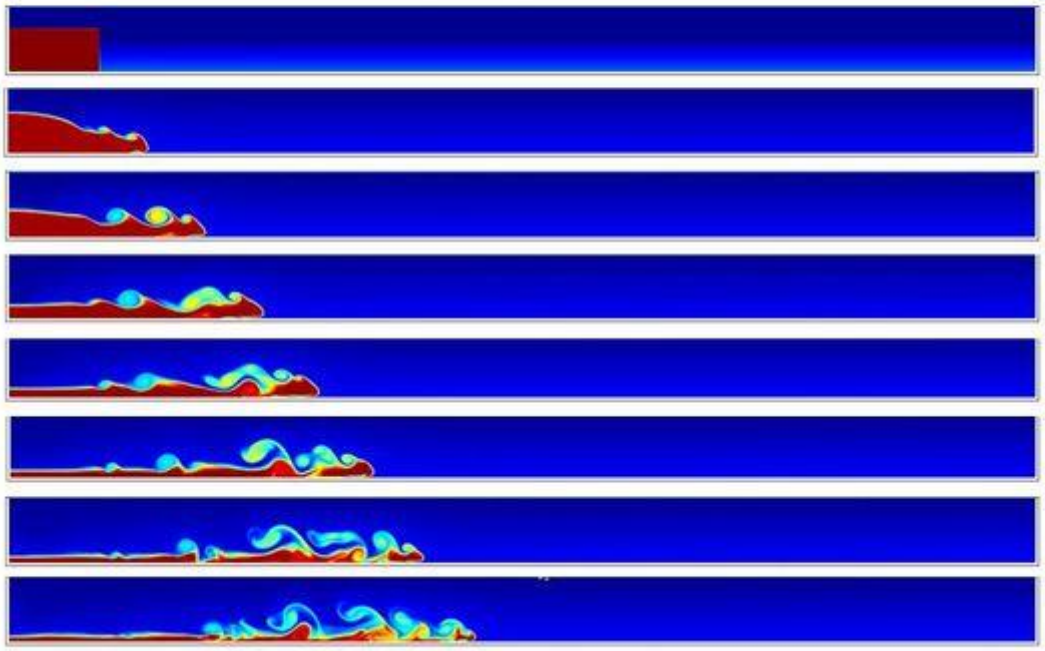


Figure 9.11. Gerris results at 0.9 s steps. The panels coincide with those from Figure 10.9.

O'Callaghan et al. (2010)

The final simulations that we wish to reproduce are from O'Callaghan et al. (2010) (hereinafter O10). We use the same procedure as before; we want to match the published figures of flow using as much data from the paper as possible. This is more difficult than it sounds for this paper because there are discrepancies between the description of the simulations and the results. I will discuss some of these inconsistencies before attempting to devise experiments to test our understanding of what was actually done.

O'Callaghan simulated a tank with dimensions 0.2×2.4 m using 12 GFS boxes. This is the same as the experiments described by Hartel et al. (1997). The gate between dense and light fluids is

at the center, which limits the computational domain to 1.2 m. The right side represents a gravity flow at the seafloor and the left side simulates a surface plume of less-dense water.

- They state that Pr ~ 7 for all simulations because they used $\nu = 10^{-6}$ m^2/s and $D = 1.4\times10^{-6}$ m^2/s. The Prandtl number, Pr = ν/D (ratio of kinematic viscosity to diffusivity) is then equal to $10^{-6}/1.4\times10^{-6}$ or approx. 0.72; the value for water is 7. It is possible that $D = 1.4\times10^{-7}$ m^2/s, which is the thermal diffusivity of water. The actual input parameters for the GFS simulation file are: the dynamic viscosity μ (kg/m/s) and D, the diffusivity. The thermal analog of μ is k/c_p, where k = thermal conductivity (W/m/K) and c_p = specific heat capacity (J/kg/K). When these two material properties are combined the units are (J/s/m/K)/(J/kg/K) or (kg/m/s). This thermal analog to the dynamic viscosity μ apparently has no formal name, which may have confused them as much as it has me. The model input thus requires *different* units for these parameters, perhaps because the viscosity appears in the heat transport equation whereas Pr is used to represent thermal diffusivity. Density is used to normalize viscosity for the dimensionless Navier-Stokes equations.
- They introduce Gr as defined in Equation (9.3) above for their initial tracer distribution per Härtel et al. (1997). I do not know whey they did this since Gerris has no problem with a discontinuous distribution as did the wave-solution model from the prior study. This means that h is the half-channel width, not the gravity wave front height. Furthermore, they state that $\sqrt{\text{Gr}} \sim \text{Re}$.

They define u_b (buoyancy velocity) as in Equation (9.3), and $\text{Re}_f = u_f \cdot h_f/\nu$ as described above and following Härtel et al. (1997).

- They refer to their Table 1 for the values of Re (UL/ν), Gr, and ν that were used in their test cases. This implies that the value of Re is not Re_f because U and L are traditional names of nondimensional variables. In this case, Table 1 makes no sense. For example $(3.13\times10^5)^{1/2} \sim 600$ (559.5 actually), not ~300 as reported in row 1 of the table. The other rows have similar discrepancies. Where does this factor of ~2 come from? This table is only referred to at this point. I think this section was just inserted with little or no proofreading.
- They justify the use of a slip boundary condition as being appropriate for transport of low-density water over high-density water, as in a surface plume. However, the boundary is not between fluids but at the top and bottom of the channel (actually a pipe). I think they mean the free surface between water and air is representable by a slip BC, and the edges of the pipe represent this air-water interface (remember it is symmetrical). Thus, the low-density water side is representing the plume.
- The discussion of $\Delta x \sim 2.4$ mm for Re ~ 10,500 is obtuse. Higher values of Re occur for either larger U or smaller ν in Gerris. The minimum cell size is only a function of the refinement level and the characteristic length L. They must have used a refinement of 9. I had to use really high refinements for high Re problems. Also, $0.2/2^y$ cannot equal 2.4 mm for any x; e.g., $\Delta x = 3.1$ mm and 1.6 mm, for $y = 6$ and 7, respectively. I think this is another typo— $\Delta x = 0.4$ mm for $y = 9$. It is possible it is something else if they intend L to be other than 0.2 m, which is their stated value.
- Of course, the minimum cell size for DNS, $\Delta x = (\text{Gr}\cdot\text{Pr}^2)^{-0.25}$ is meaningless in light of the uncertainties in both Gr and Pr discussed above.

I am not going to treat the rest of the discussion as discrepancies unless I have to. We will now proceed to reproduce their results (see Figure 9.12). The analysis of this figure requires using the caption as well as the text. In referring to the Re we must also keep in mind the multiple issues listed above. I cannot simply reproduce any of this work because insufficient information is given. To the best of my reading of the methods section, the results in their Figure 2 (reproduced here as Figure 9.12) used the following parameters: $\mu = 0.001$ kg/m/s ($\nu = 10^{-6}$ m^2/s, from which $D = \nu\rho/Pr = 1.4\times10^{-4}$ m^2/s because Pr = 7 and $\rho = 1$ (kg/m^3) for a nondimensional simulation. Note the discrepancy here; D is given as 1.4×10^{-6} m^2/s in the methods section (κ and D are synonyms).

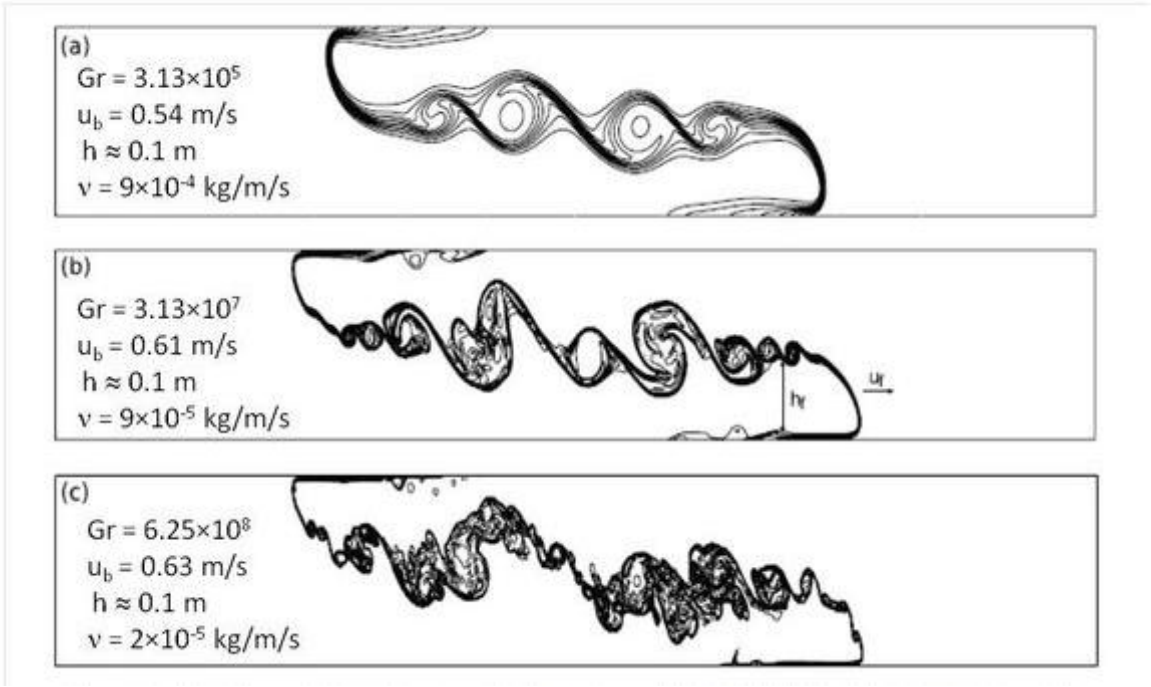


Figure 9.12. Reproduction of Figure 2 from O'Callaghan et al. (2010).

All of the prior studies used Gr as defined in Equation (9.3) to compare results. This has been useful because it can be computed relatively easily from published results. This has made this report feasible when other input parameters were unavailable. However, O10 use Re to relate their simulations and used the approximation, $Re \sim Gr^{0.5}$ instead of $Re_f \sim 1.1Fr_f{\cdot}(Gr^{0.5})$ (Hartel et al., 2000).

Figure 9.12A is for Re = 300, for which O10 report $Gr = 3.13\times10^5$. The flow should be intermediate between flows for $Gr = 10^5$ and 6.125×10^5 (Härtel et al., 1997) (Figures 9.4C and 9.4D). In fact, it is similar to these as well as the NRL experiments for $Gr = 6\times10^5$. The result for Re = 2600 ($Gr \sim 3.13\times10^7$) is similar to results from Härtel et al. (2000) for $Gr = 4\times10^8$, as well as the NRL simulation for this flow. The nose result for $Gr = 4\times10^8$ from our simulation does not reveal as much structure as O10s simulation (Figure 9.12C) for Re = 10,500 ($Gr \sim 6.25\times10^8$) but it is very similar.

The other issue we need to address with respect to Figure 9.12 is the propagation velocity, $u_b = (g' h)^{1/2}$, estimated from the model results. This is important because h can be defined in different ways. Another discrepancy in O10 comes out here. They report $u_f = 0.54$ (Re = 300), 0.607 (Re = 2600), and 0.625 (Re = 10,500) for their simulations (Figure 10.12). The approximate values of Gr for these simulations are 10^5, 10^7, and 10^8. The values of Fr_f from H00 for these Gr are 0.51, 0.6, and 0.63, respectively. It seems that O10 are referring to Fr and not u_f in their discussion of these simulations. We can check this with our results as well.

We can compare some of our simulations (Figure 9.13) to these results to understand this relationship better.The first simulation (NS-2a) uses a large viscosity (0.01 kg/m/s) and small ρ' = 0.1% to get Gr = 9×10^4 (Re ~ 300). The gravity flow front propagates at $u\_f = 0.016$ m/s and $u_b = 0.03$ m/s. The resulting $Fr_f = 0.54$ ($h = 0.1$ m), which is slightly high for this flow. Simulation NS-1 has a smaller viscosity and a ρ' of 6.3%; thus, $u_b = 25$ cm/s. The resulting flow at $t = 6$ s (Figure 9.12B) is very similar to the result from O10 for Re = 2600, with $Fr_f = 0.61$, very close to the data H00 (Figure 9.9). We reproduced the flow for Re = 10,500 using ρ' = 6.3% and μ = $2.38\cdot10^{-3}$. The resulting Fr_f is very close to the result form H00. These simulations demonstrate that it is likely that O10 are reporting Fr_f rather than u_f in their Table 1.

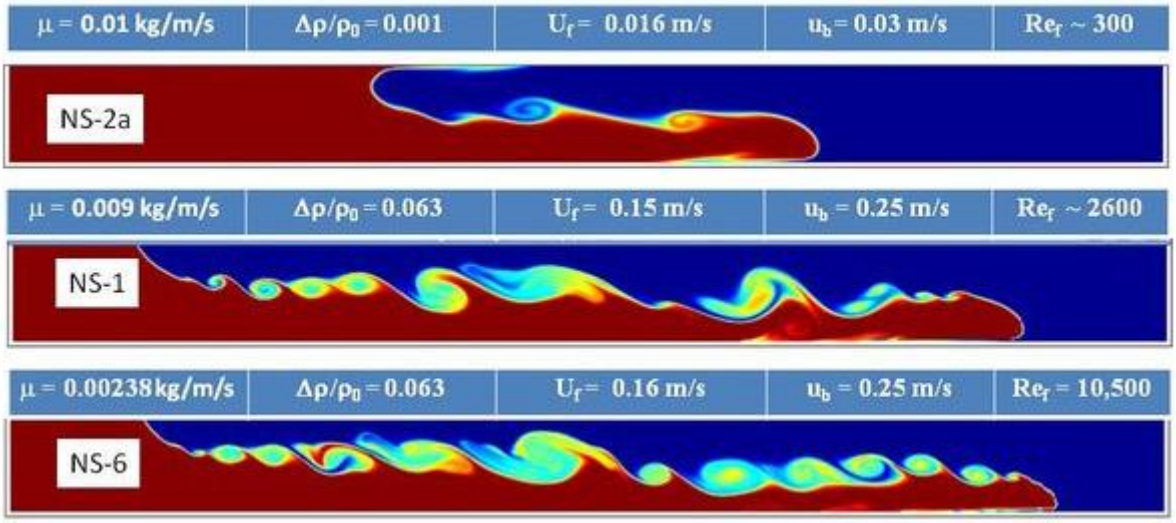


Figure 9.13. Plots of Gerris simulations at T = 11 s for low density gradients.

If the results from O10 are at $t = 11$ s, we can estimate the simulation conditions for their Re = 300 flow based on estimates and given values of parameters. The flow height $h_f \sim 0.1$ m and the total domain is 2.4 m (the flow travels at most 1.2 m). If we take Fr_f from H00 (Figure 9.9) or interpret O10's statement as a typo, we get $Fr_f \sim 0.51$. We estimate $u_f = 0.05$ m/s from travel distance and $t = 11$s. Consequently, $u_b = u_f/Fr_f \sim 0.1$ m/s. We can then use the relation for $u_b = (g' h_f)^{1/2}$ to estimate ρ' as 0.01. Their higher Re flows were limited by adjusting the flow parameters to keep them in the same region of the domain.

Summary

The lock-exchange simulations were intended to make certain that I understand the basic requirements for using density variations and interpreting the simulations with respect to nondimensional CFD principles. The primary question that has come up is, *When should reduced gravity be used?*

Simpson and Britter (1979) introduce the Boussinesq approximation to reduce their experimental variables (ρ_1, ρ_2, g) to $g' = (\rho_2 - \rho_1)g / \rho_1$, because their fluids have small density differences; $(\rho_2 - \rho_1)/ \rho_1$ ranged from 0.0037 to 0.03. Their tank was 12 cm deep, which is comparable to those from other studies. They plot the dimensionless velocity, $Fr_f = u_f / (g' h)^{1/2}$, for several different studies in their Figure 11 but they do not list experimental values. However, we can infer typical values from their report (e.g., $0.003 < \rho' < 0.03$ and $h = 12$ cm), and thus estimate 6 cm/s $< u_b <$ 19 cm/s. Note that u_b is calculated using the total water depth rather than the gravity current height.

The conclusion of these comparisons is that we have reproduced the flow results from prior work within the uncertainties associated with the incomplete reporting of previous papers. Furthermore, these simulations were dimensional and the results are thus unambiguous. This is not meant to imply that there is anything ambiguous about the nondimensional results we have been examining. The issue is in the *reporting*, which must give complete descriptions of the physical problem because the Grashof number is sensitive to a number of fluid and flow parameters. This is exemplified in Figure 10.9 for a full-gravity simulation.

References Cited

Benjamin, T.B., 1968. Gravity currents and related phenomena. *Journal of Fluid Mechanics*, 31:209-220.

Boudreau, B.P., 1997. *Diagenetic Models and Their Implementation.* Berlin: Springer. 414 pp.

Chorin, J.A., 1968. Numerical solution of the Navier-Stokes equations. *Mathematics of Computation*, 22:745-62.

Hartel, C, Kleiser, L, Michaud, M, Stein, C.F., 1997. A direct numerical simulation approach to the study of intrusion fronts. *J. Eng. Math*., 32:103-20.

Hartel, C., Meiburg, E., Necker, F., 2000. Analysis and direct numerical simulation of the flow at a gravity-current head. Part 1. Flow topology and front speed for slip and no-slip boundaries. *Journal of Fluid Mechanics*, 418:189-212.

Lindgren, E.R. 1956. Properties of Certain Bentonite Suspensions and water—a note on the inadequate definition of the Reynolds number in hydrodynamics. *Arkiv for Fysik,* 11:117-125.

Maxworthy, T., Leilich, J., Simpson, J.E, Meiburg EH. 2002. The propagation of a gravity current into a linearly stratified fluid. *Journal of Fluid Mechanics,* 453:371-94. [3]

O'Callaghan, J, Rickard, G, Popinet, S, Stevens, C., 2010. Response of buoyant plumes to transient discharges investigated using an adaptive solver. *Journal of Geophysical Research-Oceans,* 115.

Popinet, S., Smith, M., Stevens, C.. 2004. Experimental and numerical study of the turbulence characteristics of airflow around a research vessel. *J. Atmos. Ocean. Technol.*, 21:1575-89.

Rottman, J.W., Simpson, J.E., 1983. Gravity currents produced by instantaneous releases of a heavy fluid in a rectangular channel. *Journal of Fluid Mechanics*, 135:95-110

Simpson, J.E, Britter, R.E., 1979. Dynamics of the head of a gravity current advancing over a horizontal surface. *Journal of Fluid Mechanics*, 94:477-491.

Section 10: Example Applications

Tidal Simulation in the Gulf of Maine

Introduction

This activity is preliminary for potential use of GFS to simulate the tides in the Gulf of Maine (GM). The GM is an extension of the North Atlantic Ocean, and as such it is dominated by the M_2 semidiurnal tide, which rotates counterclockwise with an amphidromic node in the central N. Atlantic (Figure 10.1). Thus, the tidal wave propagates SW along the N. America shelf and interacts with the wide platform in the GM. This interaction is complicated by the inertial frequency, $f = 2\Omega \sin\theta$, where Ω is the rotation rate of the Earth (7.292×10^{-5} rad/s). The inertial period is then given by $T_i = (2\pi)/f$. The approximate latitude of GM is 43°N and T_i is 17.54 hours. There should be very little modification of the tidal wave as it propagates through the area, in contrast to the northern Gulf of Mexico, where the inertial period is equal to the K_1 diurnal period at ~30°.

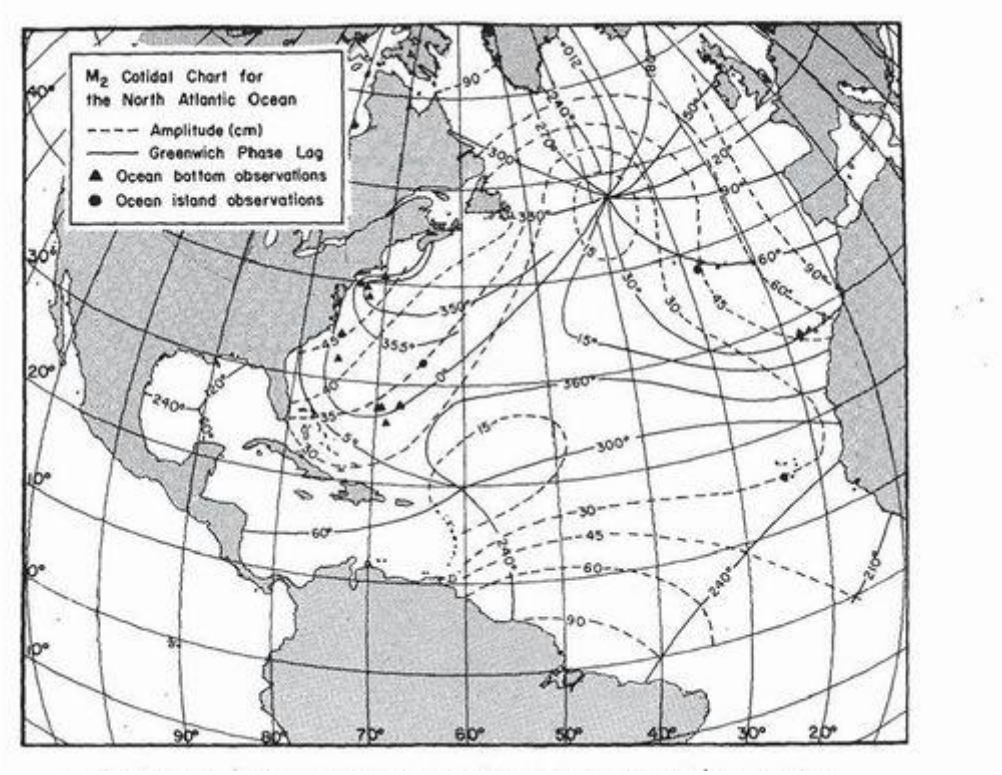


Figure 10.1. Cotidal chart of the M_2 tide in North Atlantic.

The shelf is quite large in this area because it incorporates George's Bank, however, and the M_2 tide is dissipated by passing over it. For example, the phase difference for this constituent

across the opening to the GM between stations FUNDY 21 on the north (phase = 241°) and IAPSO #30-1.2.32 on the south (phase = 347°) is ~106° or 3.6 hours. This is a distance of 600 km, which indicates a tidal propagation speed of 46.3 $m \cdot s^{-1}$. The approximate depth of the shelf is 200 m, and the theoretical propagation speed of the wave should be ~44 $m \cdot s^{-1}$, which is in good agreement with the estimate from the station data.

The large tides in the GM-Bay of Fundy (BF) region are attributed to a resonance relationship between the M_2 tidal forcing at the shelf break and the western Gulf of Maine (Brown, 1984). The cotidal chart (Figure 10.2) shows that the phase difference between the edge of Georges Bank and the upper bay is ~90°, or 1/4 of the tidal period. This mechanism has been discussed in several papers (Garrett 1972; 1984; Ku et al. 1985). Brown (1984) suggested, based on a dynamical balance approach, that the tidal wave behaved differently within different areas of the basin: (1) a progressive wave is indicated over the shallower water of Georges Bank; (2) the tide behaves as a standing wave within the western Gulf of Maine; and (3) weak progressive wave dynamics on the New England shelf. The tide must propagate over the relatively shallow Georges Bank in a consistent manner to the western Gulf of Maine, which produces much larger currents than elsewhere. This rapid propagation south of Nova Scotia is seen in the 90° phase cotidal isopleth being ~200 km advanced into the Bay of Fundy relative to the western GM. This suggests that a standing wave is generated after the tidal wave reaches the 100 cm isoline.

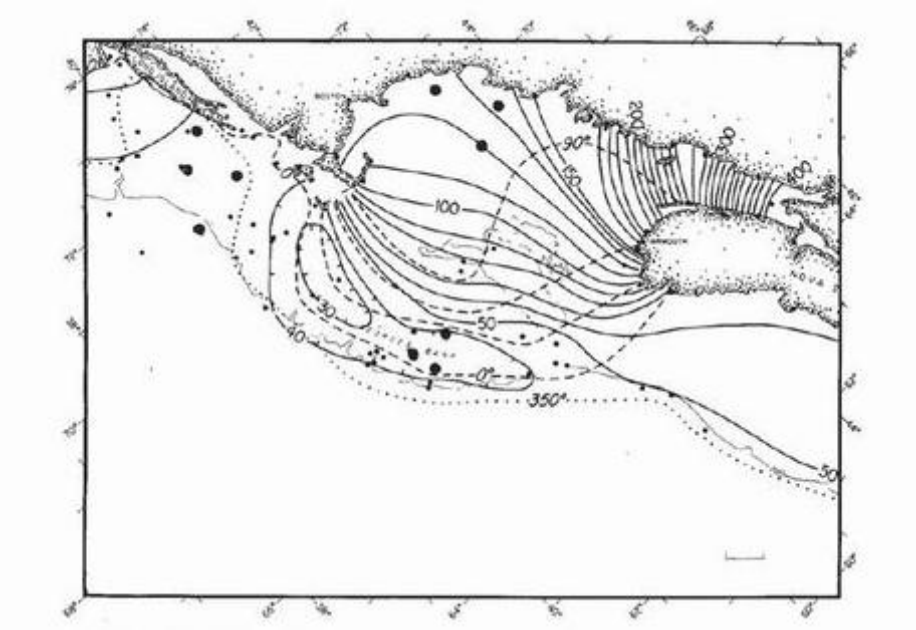


Figure 10.2. An M_2 cotidal chart for the Gulf of Maine and Bay of Fundy system.

It is further suggested by Brown (1984) that bottom friction is unimportant in the western GM because of the small bottom currents. However, he does not address the potential tidal dynamics in the Bay of Fundy. The the tidal propagation across Georges Bank is examined in detail by Chen et al. (2011) using FVCOM. This model used triangular cells that varied from 300 m to 15 km at the open boundary. They used eight tidal constituents: M_2; N_2; S_2; K_2; K_1;

O_1; P_1; and Q_1. They integrated the 3D equations for 90 days using an external time step of 12 s, and an internal time step of 120 s. Bottom stress was parametrized using a logarithmic bottom boundary layer with spatially varying z_0 except where the water depth is less than 40 m, for which z_0 was 3 mm. They do not discuss the wetting and drying formulation so it appears not to be present. This would explain why they do not discuss the upper Bay of Fundy in the paper.

Chen et al. (2011) focus on the New England shelf and western GM. They discuss tidal energy balances including dissipation. Their grid extended into the Bay of Fundy and they show high dissipation in the areas where the tide is very high. Their results further demonstrate that eddy generation by islands over Nantucket Shoal is the source of an observed phase lead for the M_2 from the slope to Nantucket Island. There is no discussion of the possible generation of a standing wave in BF.

The FVCOM model was also applied to sediment transport in the Minas Basin and Copequid Bay (Wu et al. 2011) (Figure 10.3). Flooding/drying was simulated using a mass-conserving wet/dry point treatment. The smallest cells were 100 m in Minas Basin. They used a much smaller domain, however, than previous studies. They adjusted the bottom roughness z_0 based on grain size, ranging from 0.5 to 0.0025. The model was very accurate for both elevations and currents.

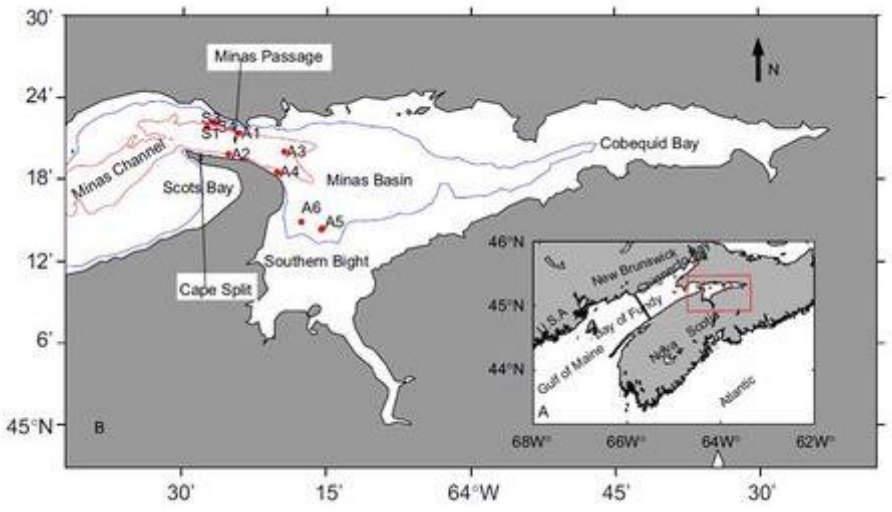


Figure 10.3. Model domain used by Wu et al. (2011) for sediment transport.

One of the most interesting results from their study was the depth-averaged residual flow, which indicates a CCW eddy in Minas Channel with a magnitude of ~1 $m \cdot s^{-1}$, and an accompanying CW eddy inside Minas Basin with weaker flow.

Objectives

This report is intended as a preliminary study on the potential use of Gerris for simulating tides in a macrotidal estuary like the Gulf of Maine and Bay of Fundy system. The scientific purpose follows from previous studies as described above. The missing part of these previous studies is

the mechanism and dynamics for the apparent standing wave (resonance) response of the M_2 tide in the Bay of Fundy. This is unique because the tide propagates over the Georges Bank and the NE margin of the Gulf of Maine as a progressive wave with significant energy dissipation. Yet, a resonance condition appears to exist in the Bay of Fundy.

The tidal behavior within the Minas Basin has also been reproduced accurately using boundary conditions near West Advocate and Minas Passage (entrance to the upper basin). Our purpose is to verify that Gerris is usable for the integrated problem and assist the USGS in implementing Gerris if they choose to. This effort will also become part of a larger study of macrotidal behavior in estuaries with substantial intertidal areas.

Methods

This study consists of three components: (1) setting up the necessary simulation conditions like the tides and bathymetry; (2) evaluating the necessary calibrations to get reasonable results from Gerris; and (3) verifying its result and completing preliminary validation using ArcGIS methods.

Simulation Description

The bathymetry is based on a data set from the WHOI THREDDS portal (http://geoport.whoi.edu/thredds/ncss/grid/bathy/gom03_v31/dataset.html). This NetCDF file was processed into an ***xyz** file, which was then transformed into a **kdt** file by the GFS utility, *xyz2kdt*. This is a terrain file used by the GfsTerrain module, which is used by the *GfsRiver* module. The general method for setting up the simulation is described for the Karamea flood tutorial (http://gfs.sourceforge.net/wiki/index.php/Karamea_flood_tutorial).

The tidal forcing at the SE and NE edges is supplied for the M_2 constituent using a constant amplitude of 45 cm and phase of 350° (Brown 1984). The tides are ramped up for 1 day. A single GfsBox centered at 42°N and 66.4°W is transformed to a Lambert Conformal projection rotated 25° CW. The box is 760 km in width. The initial refinement is 3, which is a minimum cell size of 11.87 km. A maximum refinement of 11, or 371 m, is used. The high-resolution bathymetry is melded with the Etopo1 database to fill the rotated grid completely. This is accomplished within the GfsTerrain module and requires no user action. The AMR capability in Gerris requires some adjustment to achieve the best results.

This section describes the input file, **tides.gfs**. The top of the file defines a number of C-like macros that are used when the model runs. First are some physical parameters, gravity and the Earth's angular speed. Note that M_PI is an internal constant to Gerris (π).

```
Define GRAVITY 9.80616
Define OMEGA (2.0*M_PI/86400.0)
```

Define the domain using macros for size in meters, longitude and latitude of center, and the rotation angle.

```
Define LENGTH 760e3
Define LONGITUDE -66.4
Define LATITUDE 42.0
```

```
Define ANGLE -25.0
```

Define the end time of the simulation in seconds.

```
Define ENDTIME 1728000
```

Define a minimum depth to call a cell *dry*.

```
Define DRY 1e-2
```

Define bottom drag.

```
Define CD_BOT 1e-2
```

Set max refinement level for coastline (a maximum resolution of 760e3/2^MAXLEVEL)

```
Define coastLEVEL 11
```

Set max refinement level for bathy curvature. Here weep a coarse band, 0.04 wide, on all boundaries of the domain to act as a "sponge" layer before waves exit the domain. This coarse band also helps to keep from introducing the Northumberland Strait into the simulation domain.

```
Define bathyLEVEL (fabs(rx) < 0.46 && fabs(ry) < 0.46 ? 9 : 5)
```

The following defines a simple ramp function that is applied to the boundary forcing to decrease initial oscillations. Note that *t* is a domain variable that Gerris uses. It is available for creating GfsFunctions in the input file.

```
Define RTIME 86400.0
Define RAMP(t) (t > RTIME ? 1.0 : t/RTIME)
```

The M_2 tide constituent is defined as a macro using amplitude (m) and phase (degrees Greenwich).

```
Define M2f (2.0*M_PI/44712.0)
Define M2a 0.45
Define M2p 350.0
Define M2(t) (A_M2*cos(M2f*t) + B_M2*sin(M2f*t))
```

A tide function is defined using the time-dependent amplitude (M2) and ramp function.

```
Define TIDE(t) (RAMP(t)*M2(t))
```

This is the end of the user-defined variables (macros). The simulation uses 1 GfsBox and the GfsRiver module, which solves the St. Venant (2D Non-linear Shallow Water) equations. The model simulation is given dimensions and time variables. Note that internal variables like *lon* and *lat* are assigned the defined variables from above. The two **kdt* files are listed; the terrain module will do the merging as necessary and assign the result to model variable, *Zb*.

```
1 0 GfsRiver GfsBox GfsGEdge { } {
```

Set physical length and time scales using the defined macros. Also, set the end time and max time step for the simulation.

```
PhysicalParams { L = LENGTH g = GRAVITY }
Time { end = ENDTIME dtmax = 60 }
```

Here we load cartographic projection module and set a Lambert conformal conic projection using the defined macros.

```
GModule map
MapProjection { lon = LONGITUDE lat = LATITUDE angle = ANGLE }
```

We load the Terrain module and define terrain variable (model variable, *Zb*). The basename is a list of terrain databases that will be used. These must be created beforehand and be accessible within GFS_TERRAIN_PATH. The terrain module will do the merging as necessary. We set the reconstruction of the terrain to preserve the lake-at-rest balance.

```
GModule terrain
VariableTerrain Zb {
basename = gom03_v31,etopo1_ice_g
} {
reconstruct = 1
}
```

We set some non-default advection parameters. The CFL is restricted to 0.5 for stability. Also, we choose a less dissipative limiter than the default minmod.

```
AdvectionParams {
cfl      = 0.5
gradient = gfs_center_sweby_gradient
}
```

Here we specify the Coriolis.

```
SourceCoriolis 2.0*OMEGA*sin(y*M_PI/180.0)
```

We allow the model to initialize over the first 100 steps to gradually fill and refine the coastal areas. After the first 100 steps the tide forcing is "turned on" (ramp included).

```
Init { istart = 0 } {
A_M2 = 0.
B_M2 = 0.
}
Init { istart = 101 } {
A_M2 = M2a*sin(M2p*M_PI/180.)
B_M2 = M2a*cos(M2p*M_PI/180.)
}
Init { istart = 0 istep = 1 iend = 100 } {
P = MAX(-Zb, 0.)
}
```

For convenience (useful for graphics) we define the elevation of the wet surface variable.

```
Init { istart = 0 istep = 1 } {
Hwet = (P > DRY ? H : NODATA)
}
```

Implicit scheme for quadratic bottom friction with coefficient CD_BOT is applied.

```
Init { istart = 0 istep = 1 } {
U = (P > DRY ? U/(1. + dt*Velocity*CD_BOT/P) : 0.)
V = (P > DRY ? V/(1. + dt*Velocity*CD_BOT/P) : 0.)
}
```

We refine the mesh (at beginning only) based on local curvature of terrain. The maxcells is set so that a global adaptation cost function is constructed and this "initial" refinement is not undone by the wetting/drying refinement.

```
AdaptError { istart = 0 istep = 1 iend = 1 } {
cmax = 1.0
cfactor = 4
weight = 1.0
minlevel = 0
maxlevel = bathyLEVEL
maxcells = 10000000
} (Zb <= 0 && Zb > -1500 ? Zb : 0)
```

During the simulation we refine mesh in the wetting/drying areas. The Zbn > 1 condition means refine only if the cell is coarse enough to contain at least two terrain database samples. Zbdmax is the maximum elevation of any database sample contained within the cell. H is the water elevation. The cost function is the maximum height above the local water level of any database sample. Cmax is set to zero so adaptation will go the the maximum resolution (maxlevel) whenever a wet cell contains at least one "dry" sample.

```
AdaptFunction { istart = 1 istep = 1 } {
cmax = 0
cfactor = 2
weight = 1.0
minlevel = 0
maxlevel = coastLEVEL
maxcells = 10000000
} (P > DRY && Zbn > 1 ? MAX(Zbdmax - H, 0) : 0)
```

We check for load balancing every 10 time steps (needed for a parallel run). We also print some run statistics to the screen, and output a simulation file (with all of the domain variables) that can be read by *gfsview2D*, which is part of the GFS software package.

```
EventBalance { istep = 10 } 0.1
OutputTime { istep = 100 } stderr
OutputBalance { istep = 100 } stderr
OutputSimulation { start = 0 step = 3600 } sim-%08.f.gfs
OutputTiming { start = end } stderr
```

These are the adjusted locations of some sample stations from the IHO data base.

```
GfsOutputLocation {step=900} BURNTCOAT_HEAD_ts.txt {-63.818 45.3076 -1}
        …OUTER_WOOD_ISLAND_ts.txt {-66.8043 44.5814-1 }
        …ST._ANDREWS_ts.txt {-67.0372 45.0619 -1 }
        …ROCKLAND_ts.txt { -69.0931 44.009102 -1 }
        …PORTSMOUTH_NAVY_YARD_ts.txt{-70.721 43.081 -1}
        …BOSTON_COMMONWEALTH_PIERS_ts.txt {-71.005 42.341 -1 }
        …EAST_CAPE_COD_CANAL_ts.txt { -70.486 41.774 -1 }
        …FUNDY_6_ts.txt { -67.7085 42.4494 -1 }
        …IAPSO_ts.txt { -70.8874 40.2794 -1 }
        …FUNDY_4_ts.txt { -66.8328 40.7267 -1 }
        …FUNDY_22A_ts.txt { -65.5003 42.1068 -1 }
        …FUNDY_1_ts.txt { -63.197 42.8016 -1 }
        …MILL_COVE_ts.txt { -64.059 44.4779 -1 }
        …YARMOUTH_ts.txt { -66.1355 43.744 -1 }
        …DIGBY_ts.txt { -65.7302 44.625 -1 }
        …WEST_ADVOCATE_ts.txt { -64.8202 45.3437 -1 }
} {
```

Use a second-order time integration scheme and set a minimum water level.

```
time_order = 2
dry = DRY
}
```

Define the boundary conditions for the model domain. Apply tide forcing as Dirichlet BC

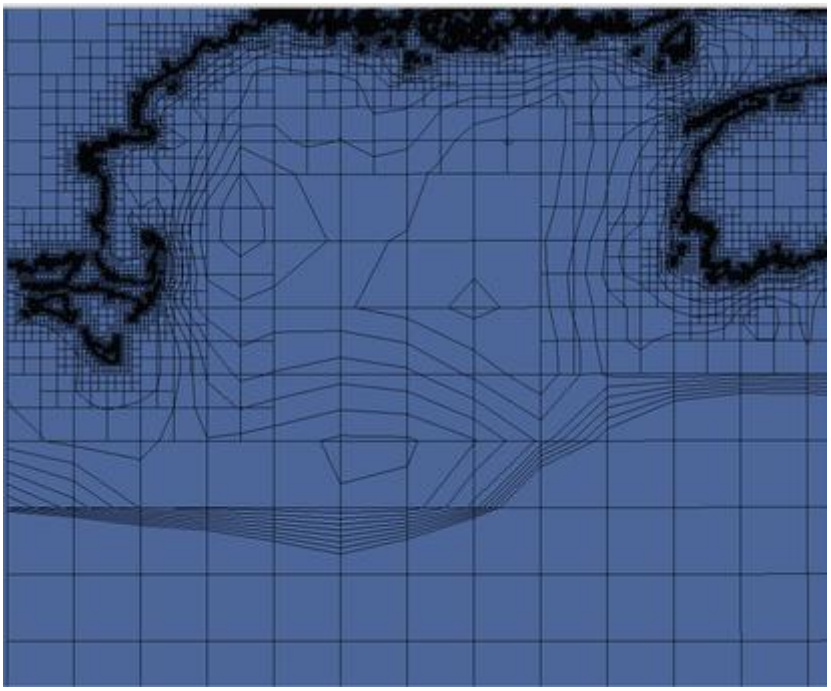
```
GfsBox {
left = Boundary
top = Boundary
right = Boundary {
BcDirichlet P MAX(TIDE(t) - Zb, 0)
}
```

```
bottom = Boundary {
BcDirichlet P MAX(TIDE(t) - Zb, 0)
}
}
```

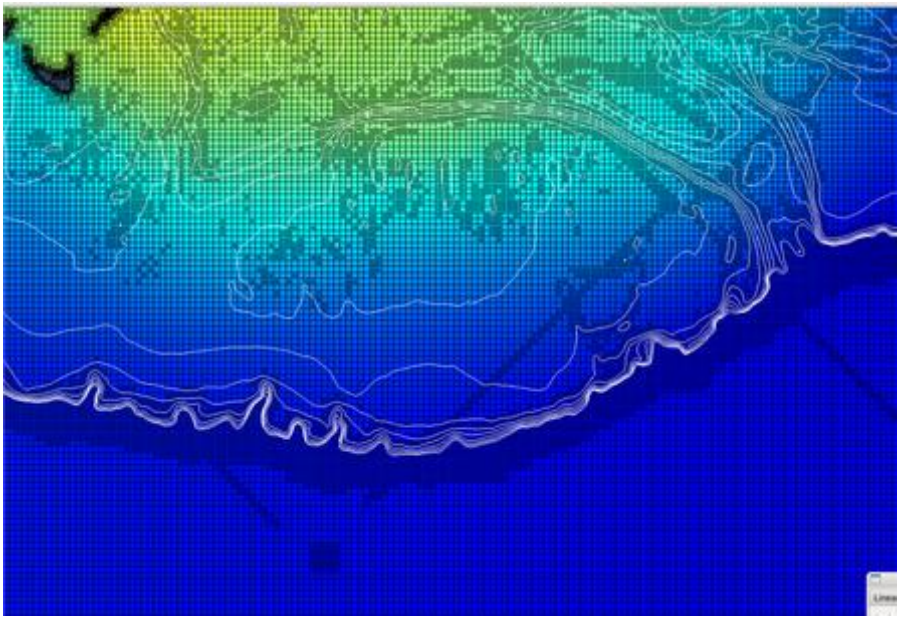
Calibrating GfsRiver (non-linear SWE)

The relevant files are located in /u/gfs/tides/gfsriver_gulf_of_maine. Within the driver script, **run_tides.sh**, there are two flags (INTERACTIVE and PARALLEL) to set the type of run. If INTERACTIVE=1, then the run will be in foreground with output piped to gfsview. If PARALLEL=1, then the run will be executed in parallel on 4 processors. The parallel run is currently set to split the gfs input (tides.gfs) to three levels (i.e., 64 boxes) and then partition the 64 boxes for a 4 processor domain decomposition. The refinement functions were adjusted to achieve the primary goal of reproducing the tides at a number of stations, which result from a resonance at the semidiurnal period. This turns out to require adequate resolution of the slope and shelf break. The run_topo.sh script can be used for generating a gfsview visualization of the terrain.

The first example (Figure 10.4A) did not refine to the bathymetry and the resulting simulation failed to produce the required amplification. This is seen in the low resolution of the shelf break. The higher refinement (Figure 10.4B) easily resolves the steep slope and produced the desired result, which will be discussed in the next section.



A. This simulation did not use the adaptError function. Contours are black.



B.This simulation used the simulation file described above. Contours are white.

Figure 10.4. Images from gfsview showing the cells and bathymetry contoured at 30 m intervals from 0-300 m depths.

Tidal Data from International Hydrographic Office (IHO)

There are a large number of tidal stations in the Gulf of Maine (Figure 10.5).

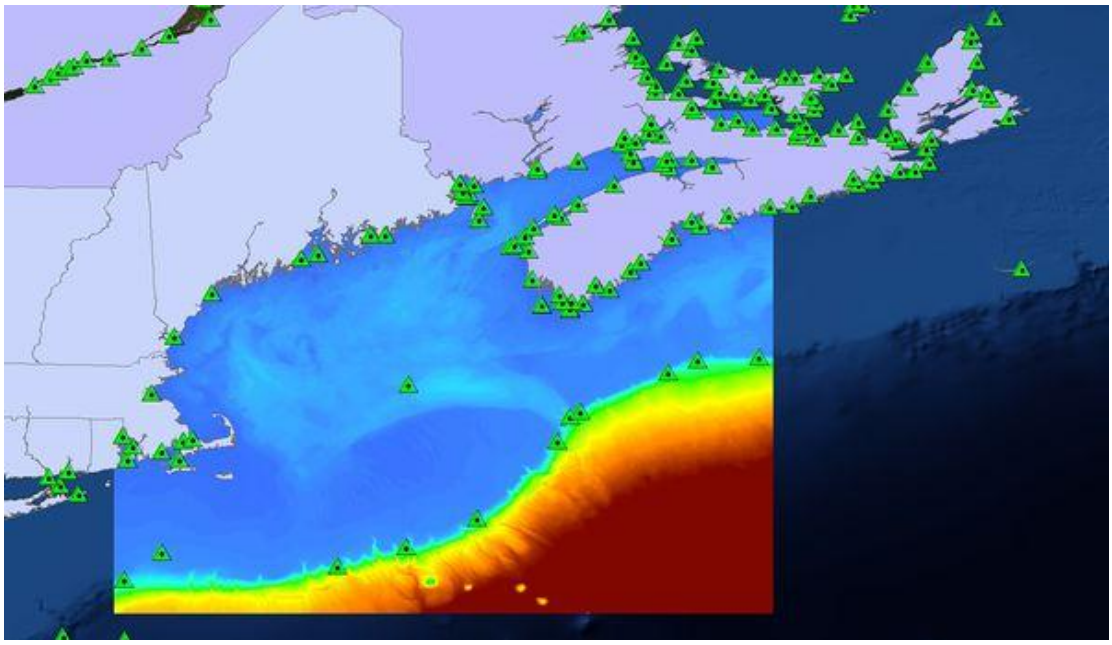


Figure 10.5. Map of 30 m bathymetry from USGS and locations of IHO stations used in this study.

We will use representative ones (Table 10.1) to evaluate the tidal elevations predicted by Gerris. They are listed here as they appear in the database.

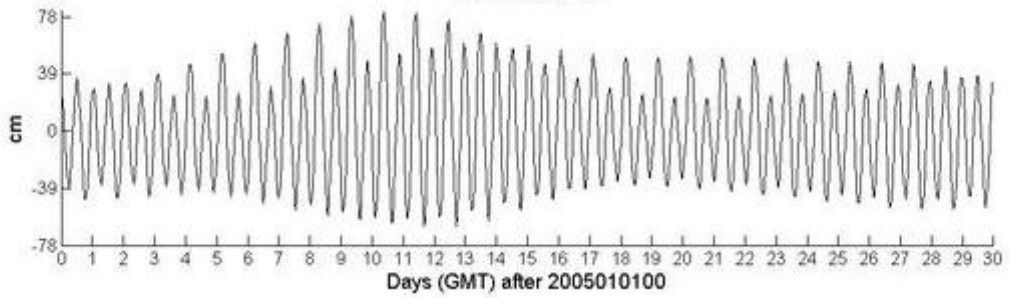
Table 10.1. Tidal Stations used for Model Evaluation

Name	East Longitude	North Latitude
BURNTCOAT HEAD	-63.787125	45.285748
OUTER WOOD ISLAND	-66.804282	44.581428
ST. ANDREWS	-67.042228	45.066838
PORTSMOUTH (NAVY YARD)	-70.725634	43.077609
BOSTON (COMMONWEALTH PIERS)	-71.020687	42.344735
FUNDY_1	-63.197	42.8016
FUNDY 22A	-65.500337	42.106789
MILL COVE	-64.063143	44.562392
WEST ADVOCATE	-64.815053	45.342856

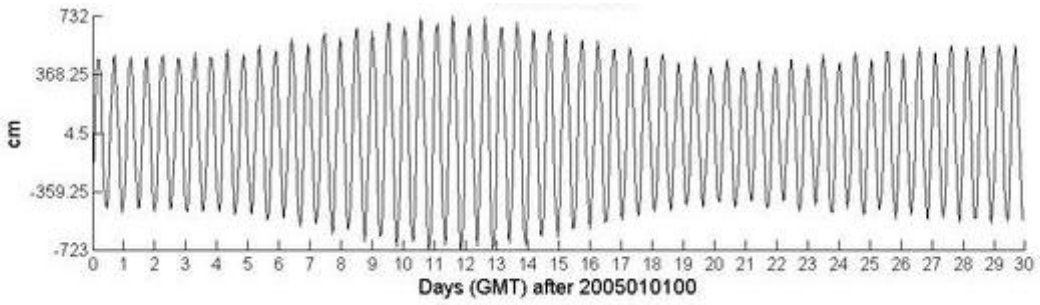
We have noted some problems in the past with respect to the locations given in the IHO database. Some corrections may be in order; for example, YARMOUTH definitely appears to be in the center of town. A better longitude/latitude for this station would be -66.138E, 43.812N. There is no simple method for assigning lon/lat values because the model results are interpolated to the requested position. In addition, there are uncertainties in assigning the water depths to the mesh at each time step because it adapts continuously. A maximum refinement can be used in specified areas of interest in addition to the method used for these simulations. The cells seen in Figure 10.4B result from refining to a large level wherever Zb is 0 (i.e., shoreline). There were difficulties in the original as well as the modified output locations. They were thus manually relocated to make sense. The new locations are listed in the simulation file above.

This study is not attempting to reproduce the detailed tidal elevations from the database. However, it is useful to identify the contribution from different astronomical forcing and the interaction between these motions. This can be examined at station FUNDY 22A (Figure 10.6). The plot shows the result of 32 constituents. We see two principle factors in the time series, however, that it would be useful to examine. The first is a fortnightly signal (e.g., spring-neap). This kind of variability is often caused by two constituents with about the same period and phase interacting. In fact the largest constituents are M_2 and S_2 with amplitudes of 45.8 and 9.4 cm, respectively. This image captures the primary signal in the region but with smaller amplitudes. If we include the other semidiurnal constituents, there is very little change. When the largest diurnal constituent, O_1 (amp = 5.5 cm) is added, we see the second visual impact; the alteration of high and low high tides every day. We note that this is not a perfectly repeating pattern, however, because the period and phase of these motions are not exact multiples.

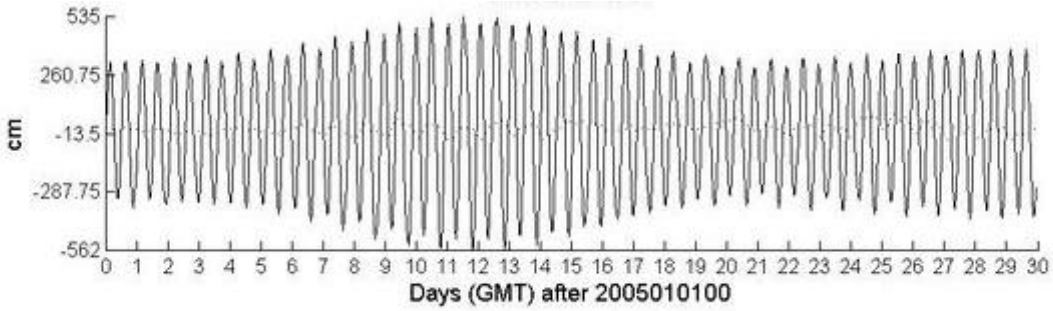
Finally, when we include the larger semidiurnal and primary diurnal constituents, the signal is very similar. These three are potential candidates for mixed-tide simulations.



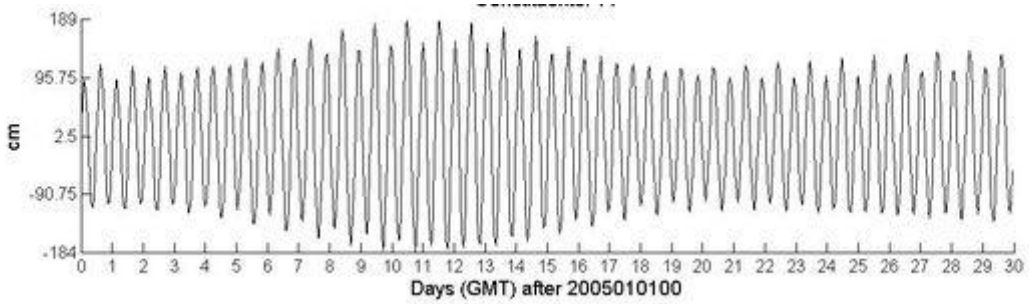
A. Fundy 22A using 32 constituents.



B. Burntcoat Head with 9 constituents.



C. West Advocate using 9 constituents.



D. Boston using 11 constituents.

Figure 10.6. Sample tidal predictions from IHO data base.

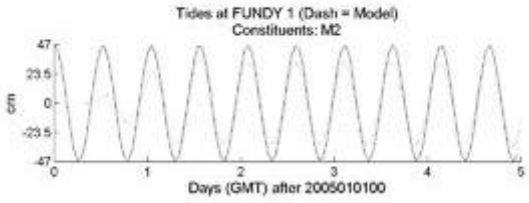
Results

We are only plotting the M_2 tides for the first set of experiments. The results can be analyzed with respect to the amplitude and phase because there are no overlapping tides.

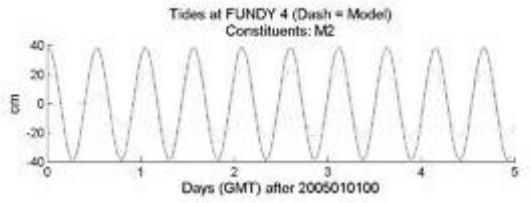
Validation

The harmonic analysis script is run in Matlab to compute the M_2 amplitude and phase as 38 cm and 236°, respectively, for the model versus 45.8 cm and 248° from the database. The next largest is the M_6, with an amplitude of 2.2 mm. This is an over-tide. The largest constituent otherwise is the S_2 with amp = 0.4 mm. The other constituents are not real in the model analysis and act as a check for the harmonic result. This indicates that the harmonic analysis is reasonable.

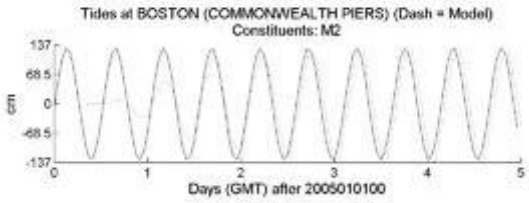
The validation for the tidal model consists of comparisons between the water levels from the IHO database and Gerris (Figure 10.7). The model has no real time so the output is shifted so that the most northern station (Fundy 1) has the correct time correlation with the database.



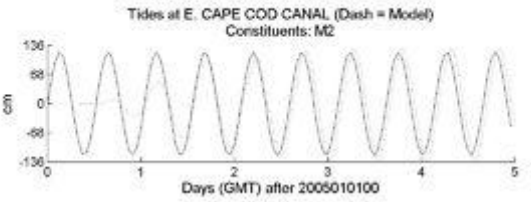
A. Fundy 1 on the NE shelf.



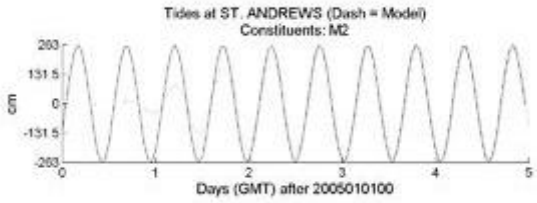
B. Fundy 4 east of Georges Bank.



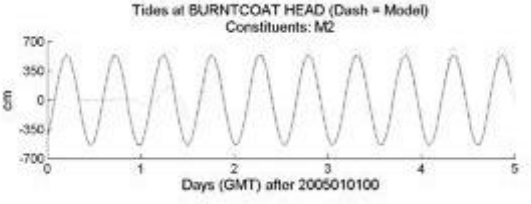
C. Boston.



D. Cape Cod Canal.



E. St. Andrews Bay in the Bay of Fundy.



F. Burntcoat Head in Minas Basin.

Figure 10.7. IHO M_2 tide time series. Gerris is shown as a dashed line.

The tidal amplitude at Fundy 1 (Figure 10.7A) is too low even though the amplitude used for the boundary condition is 45 cm. This is due to the distance from the boundary to the station. This error is more apparent at Fundy 4 (Figure10. 7B), where the tide is propagating from deep water and has obviously lost amplitude before reaching the shelf break. It is also noteworthy that the amplitude at Fundy 4 decreases with time, from a maximum at ~1.5 days. There is apparently no appreciable phase difference between these stations.

A key element of the tides in the western Gulf of Maine is the amplification of the M_2 by resonance. This is evident in the tide at Boston (Figure 10.7C), which is 130 cm. The model does have a phase error of 1.2 hr here, which could be caused by adjusting the plot to coincide with Fundy 1. The amplitude is increasing for several days, just as it is decreasing at Fundy 4. This is probably associated with an error in the amplitude at the model boundary. The tide at Cape Cod Canal (Figure 10.7D) is also slightly ahead in phase and has an amplitude that is several cms high after 3 days of simulation. This is consistent with slight errors in both the boundary condition and the bottom friction. The bottom drag coefficient used for this simulation was 0.001, which is somewhat low for shelf waters but consistent with Brown (1984); however, his results were for an analytical model and may not be appropriate. Chen et al. (2011) used a uniform bottom roughness formulation with $z_o = 3$ mm.

We can also compare the tide in the Bay of Fundy where the station location is reasonably located. For example, St. Andrews is located in a back bay at the mouth of the Bay of Fundy but the tide is very well reproduced (Figure 10.7E) with excellent phase. This demonstrates the importance of adjusting the boundary condition as well as bottom friction. It is noteworthy, however, that the tide within this region is a result of a progressive wave entering the NE side of the GOM and apparent resonance within the GOM-BF system. This is also seen in the comparison at West Advocate, where the model is very accurate with only a few minutes phase error.

The model requires some additional work, however, in Minas Basin. The predicted M_2 tide at Burntcoat Head (Figure 10.7F) is large and the model is drying out. The depth is apparently <3 m whereas the station is in a water depth of at least 5.5 m. It is not clear exactly where the station is located because the original lat/lon placed it in the town, possibly on a canal.

These comparisons demonstrate that Gerris is capturing the fundamental dynamics of the GM-BF system without detailed tuning of the boundary condition or bottom friction. It is consistent with previous results and the available data. We are not going to evaluate the currents because that is beyond the scope of this preliminary study.

Tidal flooding and drying

The most interesting aspect of this preliminary study is the simulation of wetting and drying during tides. The nonlinear SWM easily reproduces the physical mechanism but no effort has been given to using realistic bottom friction and it is evident that the available bathymetry falls short of reality. This discussion will focus on these questions. The best location to use as an example is the Minas Basin at the southern head of the Bay of Fundy. This is the area simulated

by Wu et al. (2011) for sedimentation. We will examine several locations in this area (Figure 10.8) that demonstrate the problems and opportunities that are inherent in tidal modeling in macrotidal estuaries.

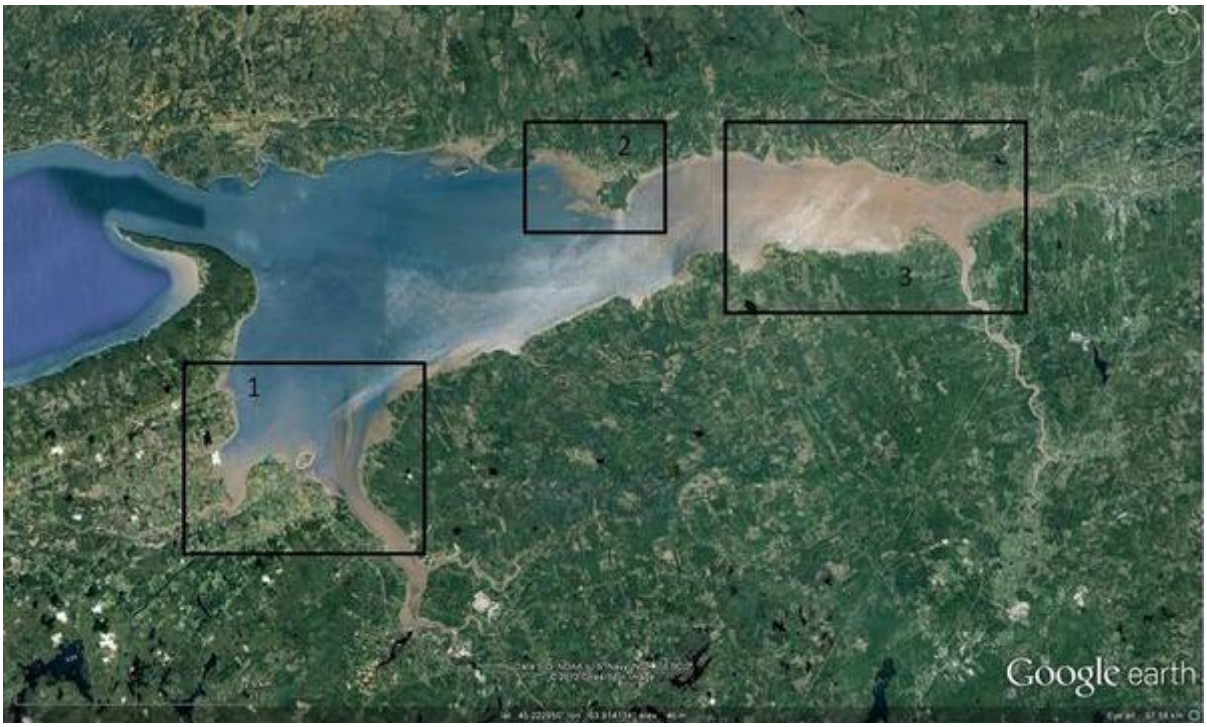


Figure 10.8. Satellite image of the Minas Basin, showing the three area discussed in the text.

We are interested in three locations from this area: (1) the SW corner of the basin where the large river enters; (2) the northern margin about half-way down the length where a headland protrudes; and (3) the narrow extremity at the end of the basin. Area (1) is occupied by farms and homes, and a delta from the river, with a permanent island and linear features intertidal and possibly subtidal features extending into the basin. Estimates of the water depth from the Google Earth data vary from -2 m nearshore to 10 m in the light-colored area where the fields are located. This is obviously not intertidal. The original 30 sec bathymetry indicates that this area is at -5 m and, consequently, Gerris predicts flooding during high tide (Figure 10.9). This area is so low in fact that it remains partly flooded during low tide, which is -5 m in this area. The model predicts extensive tidal flats around the river and the linear sand bar is emergent.

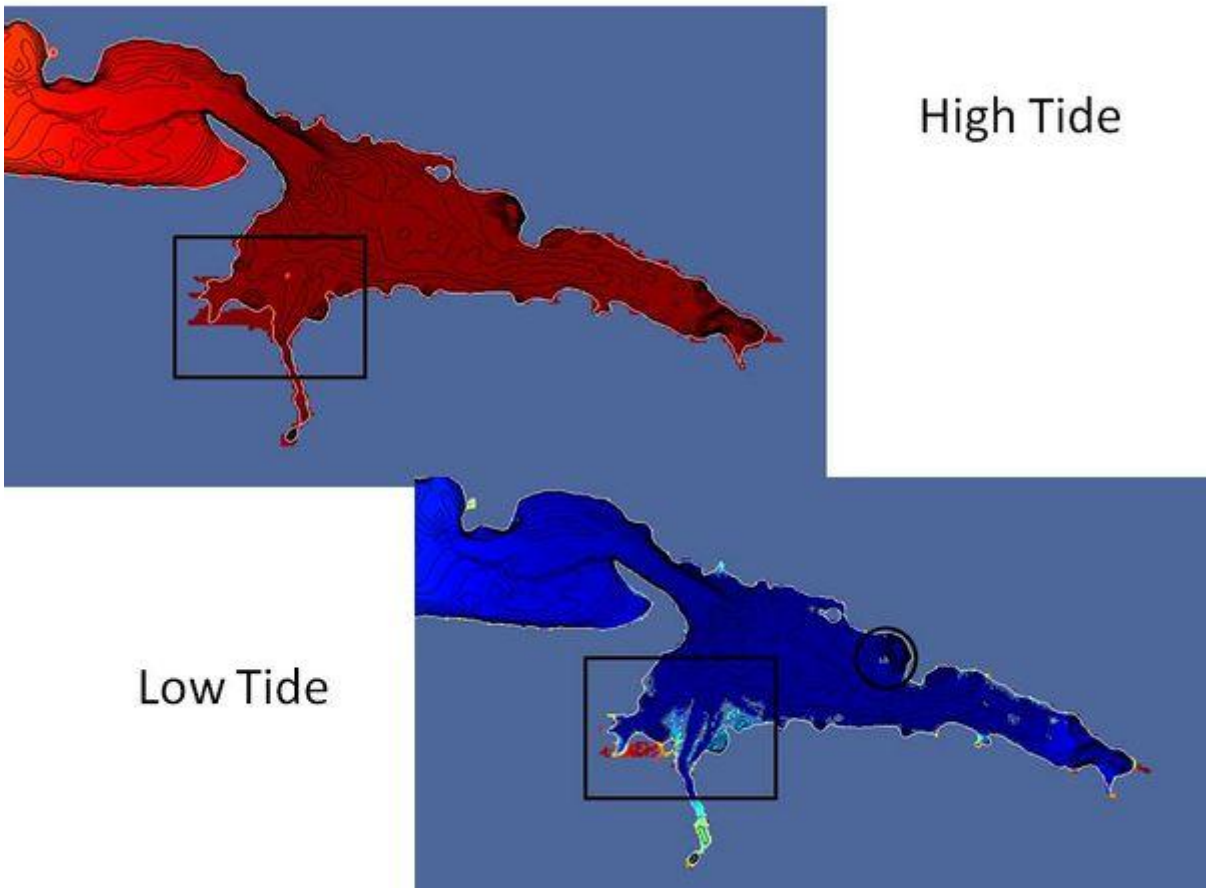


Figure 10.9. Predicted water anomaly from Gerris at high tide (upper left) and low tide(lower right). The box indicates the southern part of Minas Basin.

The second area of interest is the western margin of the north-central headland (box 2 in Figure 10.8). The satellite image shows a large light-colored area with filaments that resemble sand bars and spits associated with river flow, but there is no river at this location. The estimated elevation of this area varies from -6 m to 10 m. This suggests that it is a tidally reworked sand flat. It is evidently long-lived and appears to be eroding from the headland, which is probably of glacial origin (moraine). There is also a circular shoal to the west with an estimated depth of 2-3 m. This area is quite different in the 30 s bathymetry and in the model response. The original bathymetry does indicate the shoal, which is emergent in the model during low tide only. This is indicated in the image by the grey area that is circled. However, the bathymetry indicates a hole where the light-colored (sandy) area is evident in the image. This hole is 60 m deep; this is indicated by the closely spaced contour lines. The maximum estimated depth from the image is <10 m.

The third area is in the vicinity of Burntcoat Head (Box 3 in Figure 10.8). This area is interesting because the model predicts drying out during low tide, which is not consistent with the tidal database. The original bathymetry indicates depths of ~2 m, which explains why the model drys out with a 5 m tidal amplitude. This area is farmland on top of a rolling landscape

with a 5 m scarp at the coast. The reported location of the tide gauge is in a copse of trees 1.6 km from the coast at an elevation of 64 m. The estimated water depths do not exceed 5 m for ~2 km offshore to the north and west. The average depth of the bay is less than 5 m to the east with a narrow channel <10 m deep, whereas the 30 sec bathymetry indicates a wide channel with a max depth of ~20 m. This area also appears lighter colored, which suggests very shallow water and possibly intertidal. Of course, the actual tide level when the image was taken is unknown. The estimated depth in the bay 10 km east of Burntcoat is 30 m, which is consistent with the bathymetry. This deep basin terminates in a broad intertidal area in both databases. However, even further to the east, the 30 sec bathymetry indicates several holes up to 65 m deep. The largest of these is coincident with a depression having an estimated depth of 15 m just north of the river mouth.

Summary

This is a preliminary report on the potential use of Gerris for tidal modeling. It has demonstrated that with very little calibration the model can reasonably predict tidal elevations in one of the most challenging areas. The results indicate that the model has reproduced the resonance of the M_2 tide in the western Gulf of Maine. It has also simulated the combination of a progressive wave propagating across Georges Bank and resonance in the Bay of Fundy. The comparisons with available tidal stations indicate that complex estuaries are easily simulated with a minimum resolution of only 370 m. The dynamics of the Minas Basin, where the greatest tidal range occurs, was hampered by poor bathymetry rather than model dynamics.

We can confidently conclude that Gerris is a good candidate for tidal simulations in macrotidal estuaries. Two factors should be further investigated, however: (1) the use of a better open boundary condition; and (2) implementation of spatially variable bottom friction. We also recommend evaluating the potential use of the Ocean module because it is much faster due to its linear surface solution.

References

Brown, W.S. (1984), A comparison of Georges Bank, Gulf of Maine and New England shelf tidal dynamics. *J. Phys. Oceanogr.* 14, 145-167.

Chen, C., Huang, H., Beardsley, R.C., Xu, Q., Limeburner, R., Cowles, G.W., Sun, Y., Qi, J., and Lin, H. (2011), Tidal dynamics in the Gulf of Maine and New England shelf: An application of FVCOM. *J. Geophys. Res.* 116 (C12010), doi:10.1029/2011JC007054.

Garrett, C. (1972), Tidal resonance in Bay of Fundy and Gulf of Maine. *Nature* 238, 441.

Garrett, C. (1984), Tides and tidal power in the Bay of Fundy. *Endeavour* 8 (2), 58-64.

Ku, L.F., Greenberg, D.A., Garrett, C.J.R., and Dobson, F.W. (1985), Nodal modulation of the lunar semidiurnal tide in the Bay of Fundy and Gulf of Maine. *Science* 230 (4721), 69-71.

Li, C., Valle-Levinson, A., Atkinson, L.P., Wong, K.C., and Lwiza, K.M.M. (2004), Estimation of drag coefficient in James River Estuary using tidal velocity data from a vessel-towed ADCP. *J. Geophys. Res.* 109, C03034, doi:10.1029/2003JC001991.

Smart, G.M., Duncan, M.J., and Walsh, J.M. (2002), Relatively rough flow resistance equations. *J. Hydraulic Engineering* 128 (6), 568-578.

Wu, Y., Chaffey, J., Greenberg, D.A., Colbo, K., and Smith, P.C. (2011), Tidally-induced sediment transport patterns in the upper Bay of Fundy: A numerical study. *Cont. Shelf Res.* 31, 2041-2053.

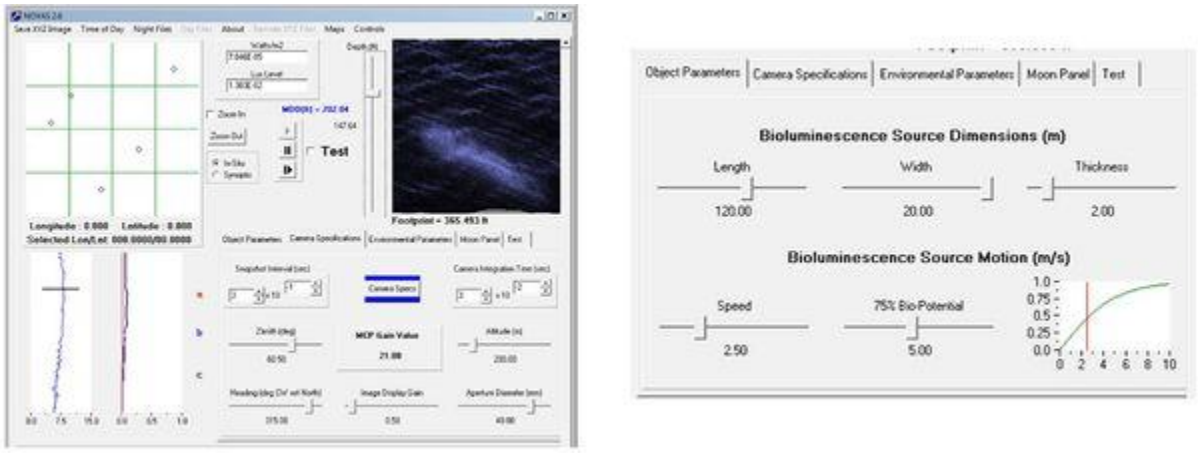
Non-Acoustic Optical Vulnerability Assessment Software (NOVAS)

To create hydrodynamical signatures of moving underwater platforms needed to derive algorithms for the associated soft 3-D signatures due to bioluminescence and bottom sediment resuspension. These signatures will then be incorporated into the Non-acoustical Optical Vulnerability Assessment Software (NOVAS) model, currently identified by NAVOCEANO as a potential operational simulation of daytime and nighttime vulnerability.

Background

Airborne detection of Navy underwater assets as they operate in the challenging littoral environment is a major concern for clandestine operations. As a submerged vehicle transits, a 3-D hydrodynamic field (3-Dhf) is generated whose characteristics will depend on platform size, shape and speed. During the daytime, sediment resuspension generated by the vehicle can become a major factor in its vulnerability as it attempts to remain as close as possible to the ocean bottom in order to avoid airborne detection of its hard signature. The inhomogeneous cloud of resuspended sediment around the vehicle, resulting from the interaction of its 3-Dhf with the bottom sediment, may increase its detectability by increasing its environmental footprint . During nighttime operations in biologically active waters, fluid shear present in the same 3-Dhf stimulates bioluminescence activity, resulting in a non-uniform distribution of blueish light that can be easily seen by an airborne observer. Because both phenomena are generated by the 3-Dhf surrounding the moving platform, it plays a crucial role in any realistic modeling of both daytime and nighttime vulnerability.

The motivation for the proposed work is best presented with a brief synopsis of NOVAS's present capabilities and needed improvements. Its interactive GUI (Figure 10.10A) allows the user to navigate through the manifold of parameters, most of which are dedicated to specifying the characteristics of the environment (wind speed for sea surface wave creation, depth profiles of absorption and scattering coefficients, depth profiles of bioluminescence potential for nighttime scenario) (Figure 10.10B) and an airborne low-light level camera (altitude, look angle, heading, aperture diameter, focal length to adjust optical zoom, and electronics).



A. Main screen for the NOVAS system.

B. NOVAS system bioluminescence controls.

Figure 10.10. Screens from the NOVAS software.

Due to its fast execution speed, NOVAS uses slider bars to quickly change the values of parameters and displays a video-like Open-GL rendering of the scene recorded by the airborne camera, for both daytime and nighttime (shown above). A realistic simulation of an airborne searching for and hovering on top of an underwater platform can be performed.

As seen from the Object Parameters tab page in NOVAS for nighttime modeling above, the 3-Dhf signature of the bioluminescent source is modeled as a very cartoonish ellipse of variable length, width and thickness. In addition, the percent of bioluminescence radiated is a function of platform speed, modeled as a hyperbolic tangent with adjustable curvature through the 75% Bio-Potential slider shown above. The upper and lower limits of the curve seem reasonable, as an asymptotic behavior for the percent of bioluminescence radiated is expected when the platform speed becomes significant. In addition, due to this simplistic 3-Dhf signature, NOVAS does not presently have the capability to model the associated bottom sediment resuspension phenomenon in littoral waters or the platform's wake. The increased realism of the NOVAS model resulting from this proposed effort will provide the Warfighter with a better awareness of his/her vulnerability, as well as an appreciation of the limitations and constraints imposed by the challenging littoral environment during the execution of a covert mission.

Objectives

Of central importance to incorporating these improvements into NOVAS is the generation of the 3-Dhf soft signature for a particular platform and to characterize its interaction with both a bioluminescing environment and a nearby ocean bottom where sediment resuspension can be initiated due to coupling with the 3-Dhf soft signature. Some obvious questions that arise for the nighttime (N) NOVAS scenario of bioluminescence are: N1) for what scenarios is the relationship between platform speed and percent bioluminescence potential a good approximation? N2) can a better characterization involving percent bioluminescence potential

as a function of shear stress be developed and quantified? N3) can the bioluminescence signature from the hydrodynamical field due to the propellers be quantified as well? N4) how does bottom sediment resuspension affect the vulnerability/detectability of a bioluminescing platform? The daytime (D) NOVAS scenario begs answers to additional questions: D1) how does the sediment resuspension alter platform vulnerability under various bottom compositions? D2) how does platform speed affect the minimum separation needed for the generation of bottom sediment resuspension? For both nighttime and daytime (ND) NAVO scenarios, questions to be addressed are: ND1) are there operating conditions in which vulnerability is reduced; ND2) how accurate must an object flow model be to provide optical vulnerability assessment, and ND3) can vulnerability be assessed in real-time with available information?

General approach

The key milestones needed for successful accomplishment of the stated objectives are:

1. Run preliminary simulations with Gerris for the case of a simple solid (prolate spheroid) to estimate computational requirements for the projected effort.
2. Generate autoCAD models of AFF1 (bare) and AFF8 (fully appended) hulls and save them in format compatible with Gerris (STL).
3. For each autoCAD model, run Gerris at different vehicle speeds, altitudes, angles, and types of sediment to produce and save the corresponding 3-Dhf lookup tables.
4. Develop an algorithm to obtain a 3-D bioluminescence footprint from any of the 3-Dhf signatures.
5. Develop an algorithm to obtain a 3-D distribution of resuspended sediment from any of the 3-Dhf signatures.
6. Convert the 3-D distribution of resuspended sediment to a 3-D distribution of inherent optical properties.
7. Improve NOVAS's raytrace algorithm to sample the predicted 3-D bioluminescence and include the 3-D distribution of inherent optical properties into NOVAS's radiative transfer module to provide a more realistic vulnerability assessment;
8. Validate NOVAS's predictions against field data.

Numerical Modeling

Direct Numerical Simulation (DNS) of turbulent flow past a vessel hull (Figure 10.11) has become possible in recent years with the advent of modern high-performance computing and adaptive grid, parallel computational fluid dynamics (CFD) models. For detailed turbulence calculations that affect the performance of the hull, these methods are still not in common use (e.g., Alin et al., 2010) but they can be used effectively for environmental computations (e.g., Popinet et al., 2004). Previous work with Large Eddy Simulation (LES) and Reynolds-Averaged Navier-Stokes (RANS) models has indicated problems with the closure schemes used for turbulence near the hull. For more complex hull designs, such as including a fairwater and

fins, the LES produces better representation of secondary flows. It is expected that DNS will produce even better results for the pressure distribution.

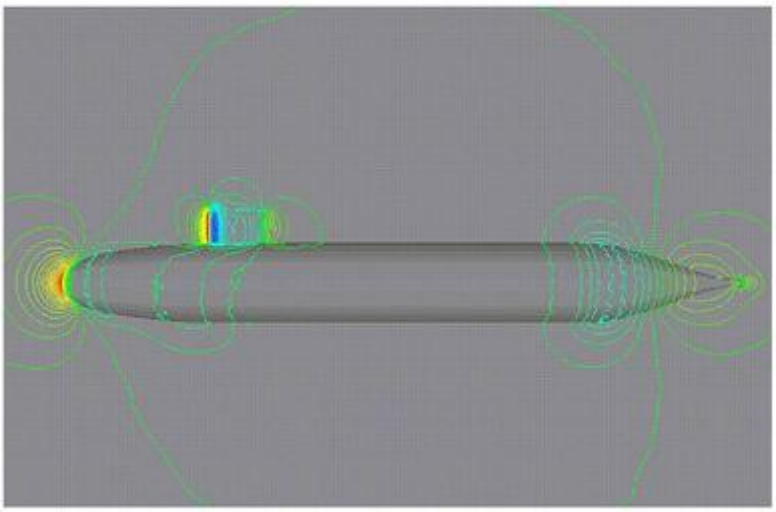


Figure 10.11. Pressure field computed for a standard hull configuration using the SUBOFF program.

We propose to adapt and integrate a 3D adaptive-grid Navier-Stokes model called Gerris, which is being used to study flow in estuaries in a current 6.1 project at NRL-Stennis, to produce 3-Dhf signatures of moving platforms and ingest them to existing bioluminescence simulation and sediment resuspension models to produce soft-body signatures. The Gerris code was developed to analyze flow around solid objects, which it reads from a standard CAD file format. This will allow multiple objects (e.g., the vehicle and the seafloor) constructed using an external program to be placed in the flow. For example, CFD analysis has been validated in designing submarines and towed underwater vehicles (Lee et al. 2003; Wu et al. 2005). We expect to be able to reproduce in-house the results shown below from a code called FEFLO (Finite Element FLOw solver) that was developed over 10 years ago at the Laboratory for Computational Physics and Fluid Dynamics for the SUBOFF program.

However, the CFD needs for the proposed work are relatively simple and the computations can be completed with either FEFLO or Gerris. We chose the latter because we are familiar with Gerris and using it now for similar purposes and we are also going to be looking at pressure effects on the seafloor and resulting sediment resuspension (Keen). The Gerris code, which is a part of the Gnu Flow Solver (GFS), is undergoing continuing development and support for a range of applications (Popinet 2003; Popinet et al. 2004; Rickard et al. 2009). The submerged platform shape will be designed with available specifications using either a GNU or commercial CAD program. The CFD model computes the variations of the currents and pressure field around the object. The movement of the platform will be simulated by an upstream boundary condition representing the vehicle's speed. For example, a Volume-of-Fluid (VOF) surface can be inserted as to examine water surface disturbance as a function of depth of the object (Popinet 2009). In addition to changing its speed, the vehicle can also move within

the material (e.g., changing depth). A bottom surface will be inserted to predict the impact of the flow field on sediment resuspension (Tang and Keen 2011). Furthermore, the independent interaction of control surfaces with the flow can be examined (Lee et al. 2005). This will permit a range of detail in examining the pressure field under realistic ocean conditions in 3D using the TecPlot visualization program.

Preliminary experiments will represent a submarine hull with a prolate spheroid (6:1) at high Reynolds number in a 2d axisymmetric flow simulation. This is a common first-step in studying this problem (Givler et al., 1991). Although the focus of the proposed work will involve a towed hull, we will also explore and evaluate the possibility of modeling the propeller wake's 3-Dhf.

Numerical experiments

We will acquire either the specifications for the AFF1 (bare hull) and AFF8 (fully appended model) hulls used in the SUBOFF experiments completed with DARPA funding. Because of the expected high speeds of the submarine, these numerical simulations will be completed at Reynolds number >1×106. We will complete base experiments to determine the required degree of refinement of the numerical grid for the expected large number of simulations required for the look-up table product. The simulations will include increments of hull speed as well as angle. It is expected from previous work that the greatest turbulence and resulting pressure variations will result from changes in hull angle during maneuvering. The experiments will be scaled for either the SDV or a submarine. Any variations from this nondimensional result because of the unusually slow SDV speed will be accounted for by extending the lower Reynolds number bound.

In addition to the base experiments with the AFF1 hull, some cases will be run with the more complex AFF8 hull in order to estimate the error associated with the simplified hull. It is expected that the computational requirements of the fully appended hull will restrict its use somewhat. The exact availability of these simulations will depend on the outcome of preliminary results. The experiments with the AFF8 hull will use the 3D Gerris solver for ½ of the hull (bilateral symmetry) to reduce computational requirements.

The experimental setup will include steady flow in a wall-bounded channel with uniform temperature and salinity. The bare hull simulations will be axisymmetric 2D with a sea bottom represented by a no-slip boundary condition at the "bottom" of the channel and a free-slip condition for momentum at the "top". Pressure fluctuations at either boundary resulting from turbulence generated by the hull will be used to compute either sea surface variations or sediment resuspension at the bottom. The suspension of sediment will be parameterized for several classes of materiel, including sand, clay, and organic detritus.

Processing input

Once an autoCAD model has been created for a particular platform, scripts will be written to generate input files for Gerris in order to automate the process of running Gerris for different vehicle speeds, altitudes, angles, and types of sediment, as well as to save the Gerris results in the form of 3-Dhf lookup tables needed for Milestones 4) and 5).

Create bioluminesence results

The 3-Dhf produced in Milestone 3 will be converted to 3-D bioluminescence soft signatures via an algorithm that relates percent of bioluminescence radiated as a function of fluid shear, instead of platform speed as is presently done in NOVAS and explained earlier. Although fluid shear is expected to be proportional to platform speed, this perhaps intricate relationship can be directly by-passed with a direct conversion from fluid shear to percent of bioluminescence radiated. On-going collaboration with Mike Latz from Scripps Institution of Oceanography over the last few years will allow importing the results of his group's efforts into NOVAS. To quote from one of his group's recent annual reports (Latz et al., 2010): "once a transfer function between the flow agitator and flow field is known, it can be used with the NAVOCEANO METOC database of bioluminescence potential measurements to predict bioluminescence signatures in essentially any oceanic region. The Non-acoustical Optical Vulnerability Assessment Software (NOVAS) being developed ... has a placeholder in which the coupled BIOSTIM-CFD model can be incorporated into the nighttime visibility assessment component."

Create SPM fields

The 3-Dhf produced in Milestone 3 will also be converted to a 3-D distribution of resuspended sediment by leveraging an in-house model (Keen) that presently predicts the 1-D depth profile of sediment that is resuspended due to bottom currents. The algorithm will be extended to predict a 3-D distribution that will be needed for Milestone 6.

Create IOP fields

The 3-D distribution of resuspended sediment from Milestone 5 will be converted to a 3-D distribution of inherent optical properties. An in-house algorithm developed by Haltrin to perform this conversion for a 1-D depth profile of resuspended sediment will be leveraged to reach this milestone.

Extend ray-tracing algorithm

Due to the simple cartoonish representation of the platform in NOVAS discussed previously, its present raytrace only interrogates the water column depth at which the platform is located. The raytrace routine will be extended to interrogate all the layers of the water column in order to sample the 3-D bioluminescence soft signature. The 3-D distribution of inherent optical proper property will be ingested into NOVAS's radiative transfer module.

Validate results

Data on Swimmer Delivery Vehicle vulnerability and bioluminescence signatures is currently being collected by NSWCCD and is a potential source of validation data for the proposed bioluminescence modeling. In addition, other on-going work with JHU/APL (George Klaus) funded by NAVOCEANO N9 are underway to assure such data are available. Past data collection of low light signatures from JMMES or the NRL camera system are also available and will be used to provide validation of the predicted vulnerabilities.

Modeling approach with Gerris

Initial NOVAS tests for a hull in a steady flow have been completed and are reported in this section. The top directory is /home/keen/PROJECTS/NOVAS/GERRIS on typhoon. The project is called NOVAS.

Swimmer Delivery Vessel (SDV) represented by an ellipse in a single box with the following model characteristics

- 30 m box
- GfsSolid { ellipse {0, -0.5, 0.33, 0.1}} = 10 m × 3.3 m
    - m/s = 0.033 box/s
- L = 30 m
- U = 1 m/s
- T = L/U = 30 s

The refinement used for the adaptive mesh is $2^9 = 512$ and the resulting highest resolution is ~57 mm. This is borderline DNS computation.

The response of zooplankton or other bioluminescence animals to the pressure anomaly predicted by Gerris can be parameterized using a simple function. This is implemented using the GfsSource function for the user-defined variable, *CHLOR* in the simulation file:

```
Define MAXTIME 50
Define PMAX  4e-3
Define PMIN -4e-3
# TPRINT is the real time step frequency to print fields
Define TPRINT 2
Define IPRINT 100
3 2 GfsSimulation GfsBox GfsGEdge {} {
Time { end = MAXTIME }
Refine 6
VariableTracer {} CHLOR
Source CHLOR { return P > PMAX || P < PMIN ? 1.0 : CHLOR; }
Init {} { U = 0.03333 }
AdaptVorticity { istep = 1 } { maxlevel = 9 cmax = 1e-2 }
AdaptGradient { istep = 1 } { maxlevel = 9 cmax = 1e-2 } P
SourceViscosity {} 0.00078125
GfsSolid ( ellipse ( 1.0, -0.5, 0.33, 0.10 ) )
OutputTime { istep = 1 } stderr
OutputPPM { istep = IPRINT }  p.ppm  { v = P }
OutputPPM { istep = IPRINT }  u.ppm  { v = U }
OutputPPM { istep = IPRINT }  chl.ppm  { v = CHLOR }
OutputGRD { step = TPRINT } u-%g.asc { v = U }
OutputGRD { step = TPRINT } p-%g.asc { v = P }
OutputGRD { step = TPRINT } chl-%g.asc { v = CHLOR }
GfsOutputSimulation { step = TPRINT } sim-%g.gfs
}
GfsBox {
```

```
left = BoundaryInflowConstant 0.03333
}
GfsBox { }
GfsBox {
right = Boundary {
BcDirichlet P 0
BcDirichlet V 0
BcNeumann CHLOR 0
}
}
2 right
3 right
```

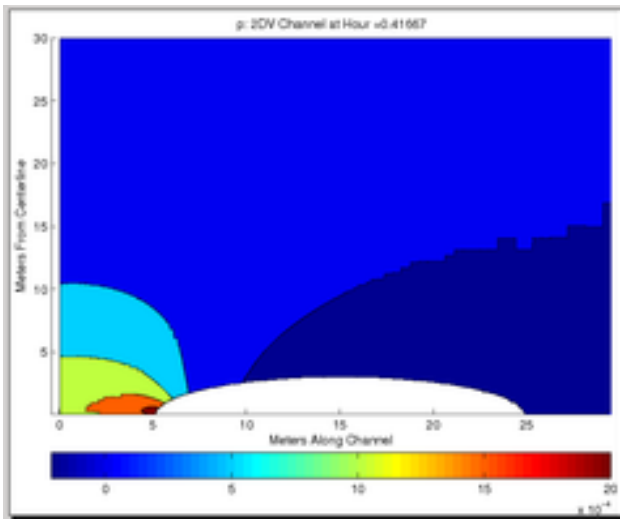
Whenever the pressure anomaly P varies 0.004 (0.4%) from the mean, chlorophyll will be created. This variable does not decay and will thus act as a wake tracer. This proxy for zooplankton is only used in the simulations with 3 boxes. It is also notable that these results are for a vessel moving at only 1 m/s. The resulting Reynolds number, $\mathrm{Re} = (d \times U)/\nu = (3 \times 1)/7.8125 \times 10^{-7} = 3.84 \times 10^{6}$, which is very turbulent flow. This implies that the pressure field will reflect a wide range of turbulent motions ranging from the Kolmogorov scale to meters.

Results

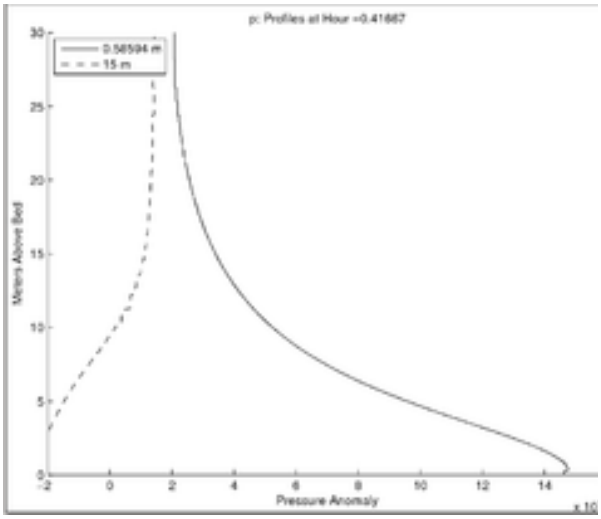
One-Gerris Box results

The first simulations use one box to represent the water around one-half of the hull. This assumes a radial symmetry. It is important to avoid interaction of turbulence with the closed upper and lower boundaries.

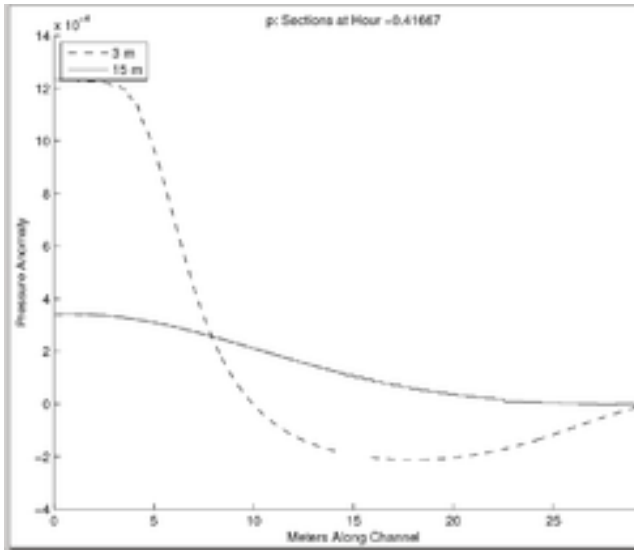
For this single box example, it is difficult to compute the flow for very long because of the limiting length. From the result (Figure 10.12), it looks like I will need either a larger L or more boxes. The pressure increases dramatically at the bow (Figure 10.12B) while a low-pressure zone is predicted from mid-length to the stern. The pressure anomaly extends to the edge of the box, at 30 m from the hull centerline (Figure 10.12C). The ambient pressure anomaly is 0, of course. This indicates that the computational domain is not large enough. The pressure anomaly is very positive in front of the hull even at 15 m from the centerline. These results indicate that this small hull (20 m length and 3 m radius) has a measurable pressure signature at 10 radii. These results cannot show the influence along its path, however. We can improve this by using more refinement and changing the flow properties to reflect this change.



A. Anomaly.



B. Radial anomaly at bow (dash) and midway (solid) from bow.



C. Axial anomaly at 3 m (dash) and 15 m (solid) from centerline.

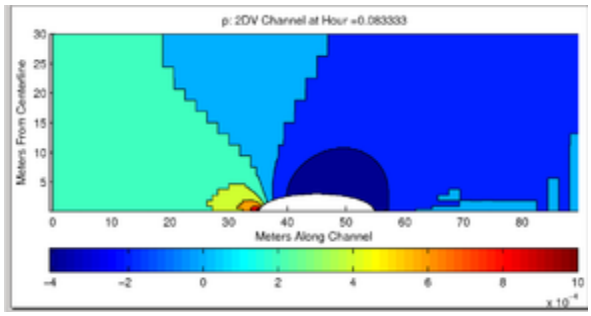
Figure 10.12. Pressure distribution for SDV hull computed by Gerris with 1 box at 0.42 hour.

These results indicate the next tests to complete:

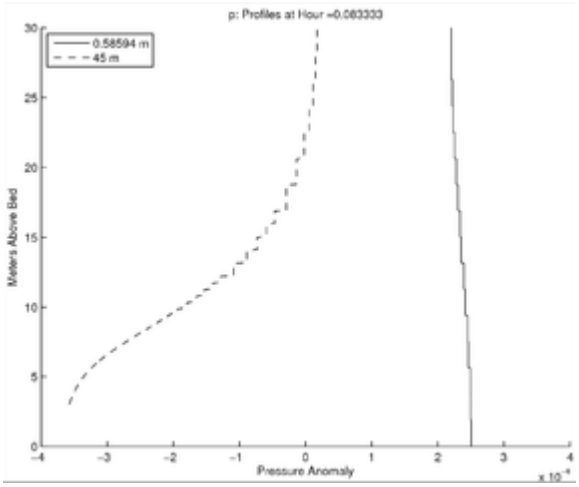
Need to find a way to propagate the effect of the pressure, either through a source of bioluminescence or a tracer of some kind that is a function of pressure.

Three-Gerris Box results

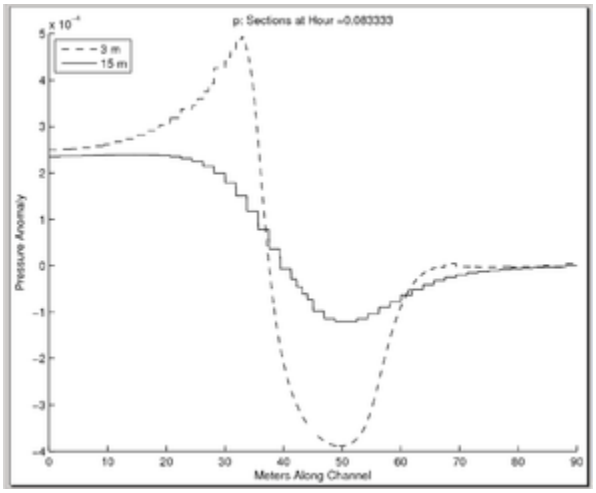
The number of boxes was increased for these simulations, and the *CHLOR* tracer was added as described above. It is important to isolate the pressure anomaly created by the hull from the boundary conditions, in order to produce robust results. The pressure field for this simulation (Figure 10.13A) does not appear to be restricted to the domain after 5 minutes, however. The radial pressure anomaly (Figure 10.13B) at the bow exceeds 2×10^{-4} at 30 m from the centerline, but it returns to ambient values ~20 m from the centerline at midway along the hull. The pressure anomaly distribution along the path (Figure 10.13C) reflects the inflow boundary condition ahead of the hull, with a perturbation of $\sim2.5\times10^{-4}$ at both 3 and 15 m. This is consistent with the result for one GfsBox above. The wake shows no discernible pressure distribution.



A. Contour of anomaly.



B. Radial anomaly.



C. Axial anomaly.

Figure 10.13. Pressure anomaly calculated by Gerris for SDV hull after 5 minutes.

It seems reasonable to treat an anomaly of 4 as a tolerance for creating *CHLOR* in the simulations (see the simulation file above). One of the interesting results from the anomaly for 3 boxes is that negative anomalies extend >15 m away from the hull at the stern (solid line in Figure 10.13C), but not so far in the wake. A wake is visible but with small values.

The results presented thus far are very likely a transient due to the flow impacting on the stagnant water surrounding the hull. The results after 48 time steps (24 minutes) (Figure 10.15) show the interaction of the flow (U,V shown as vectors) with the pressure anomaly (P contoured with a red line at $\delta P = 4\times10^{-4}$), and the bioluminescence (CHLOR contoured in black isopleths) in the figure. The interpretation of these relationships is straightforward. This would be the steady-state pattern associated with the hull moving through water at 1 m/s. The flow perturbation would be visible at the surface for water depths less than 20 m, as seen in the deflected vectors around the hull. The threshold for bioluminescence to begin is exceeded at the bow but the organisms are swept rearward past the hull. A larger threshold zone extends from mid-length to behind the stern, which further generates bioluminescence, especially near the hull where the velocity increases. These organisms accumulate at the stern and are swept into the ambient water where they will continue to be activated until they relax, at least a few meters behind the hull.

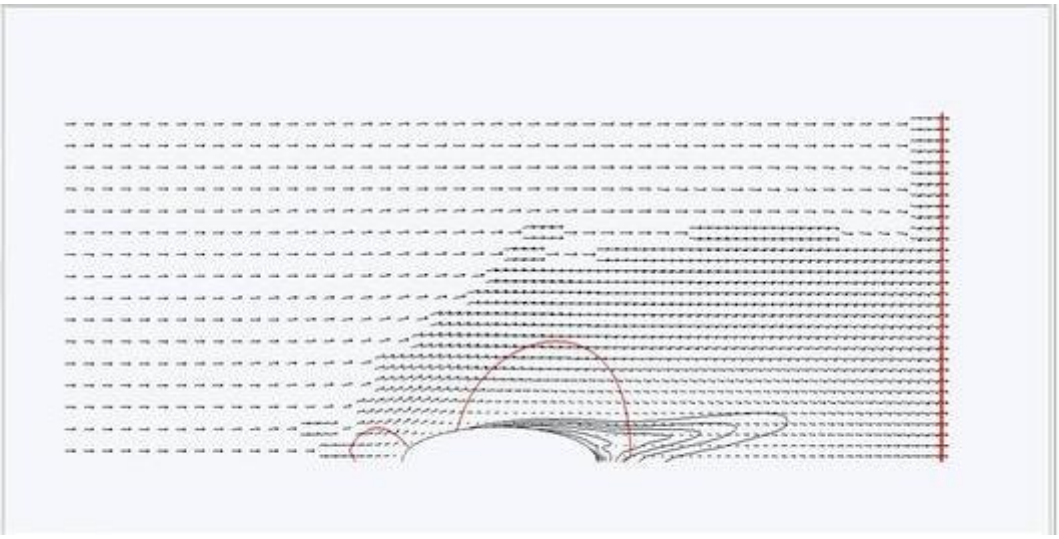


Figure 10.15. Pressure anomaly threshold (red) and bioluminescence (black) contoured over flow around an SDV hull after 24 minutes.

The bioluminescence wake is time dependent (Figure 10.16). When the flow first contacts the hull and begins to accelerate (as if the vessel were accelerating from a standstill), the pressure anomalies and resulting luminescence are at the bow. This is seen in timeseries of Relative Luminosity Units (RLU) at these locations.

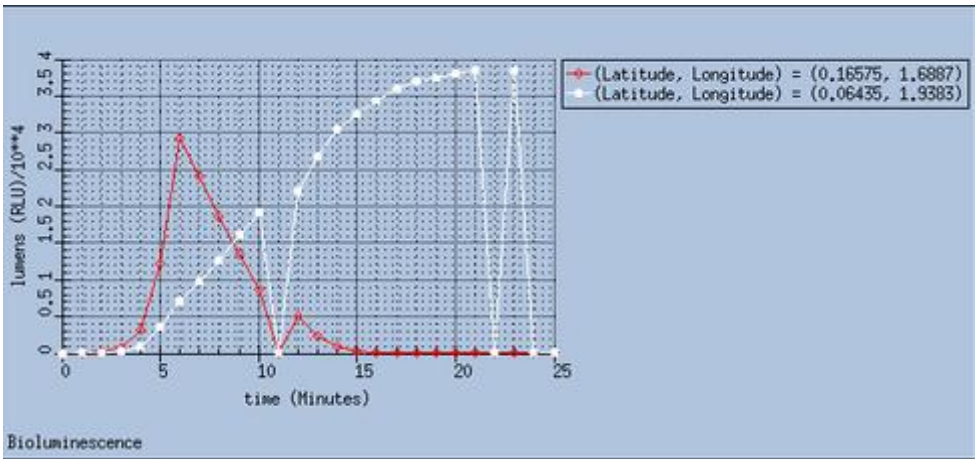
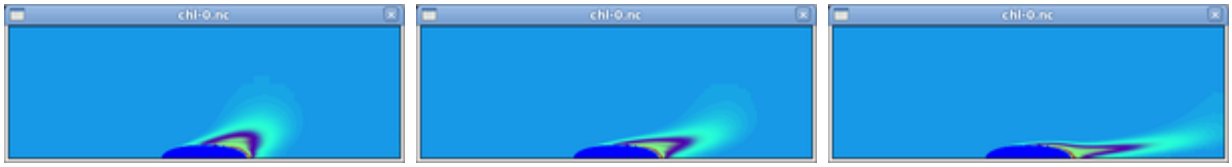


Figure 10.16. Time series of bioluminescence at forward (red) and aft (white) parts of the hull. The sudden drops are artifacts of the plotting program.

A series of plots of the RLUs (Figure 10.17) shows how the evolving flow advects the organisms as the hull reaches steady speed. The images here do not reflect the temporal response of bioluminescence organisms. It is unlikely that they continue emitting light for 25 minutes, however. This is reflected in the growing curves, which only fall off after the pressure anomaly field has stabilized.



A. 16 minutes. **B.** 32 minutes. **C.** 48 minutes.

Figure 10.17. Relative Luminosity Units (RLU) calculated from the pressure anomaly.

Instantaneous bioluminescence results

The actual response times for organisms that illuminate is variable, but typically less than 1 minute. We can simulate these organisms by not allowing *CHLOR* to be transported. For this simulation the following line was substituted for the GfsSource of *CHLOR*:

```
Source CHLOR { return P > PMAX || P < PMIN ? 1.0 : 0.0; }
```

This has the effect of resetting *CHLOR* to 0.0 whenever the pressure anomaly drops to less than the threshold. This is a reasonable approximation for organisms with response times much less than 1 minute. The reasoning is that in < 1 minute, the vessel will have traveled ~60 m at 1 m/s, which is more than its own length and any remnant bioluminescence (response time of 10 s) in its wake will be closely associated with it (~10 m). This scales inversely with hull size; for

example, a 100 m submarine hull would be equally visible for bio-response times < 100 s at this speed.

Bottom pressure and potential sediment resuspension

The hull was placed closer to the seabed in order to simulate a vessel cruising near the bottom, and the impact its flow field would have on bottom stresses and flow. This is further preparation for a second attempt at an NRL proposal. The Code 7322 tasks are defined as:

- How does the sediment resuspension alter the vulnerability under various bottom conditions?
- Are there operating conditions in which vulnerability is reduced?

The first question would involve examining environmental and vessel factors individually. The impact of different kinds of seabed material is the first factor, and the vessel hull shape, speed, and elevation are included in the second. The second question involves the relative strength of natural versus vessel sedimentation processes, which could serve as masks for vessel presence. The optical properties of the bottom material are another factor.

These simulations are based on the previous but with the hull moved up in the water column so that its lower edge is at the desired height above the bed. The first example (Figure 10.18) is for a slowly moving hull near the bed. The pressure anomaly is positive towards the front of the hull (P_{max} = 0.0004) and negative aft of mid-hull (P_{min} = -0.00167). The flow is much reduced beneath the hull as well.

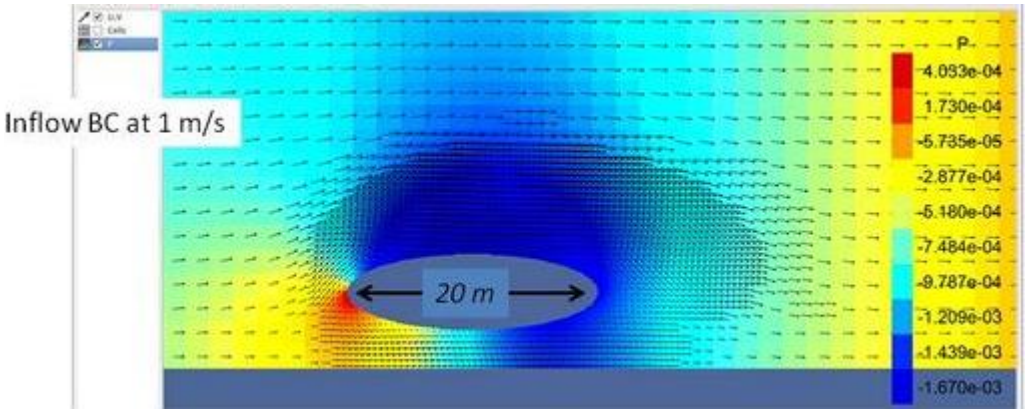
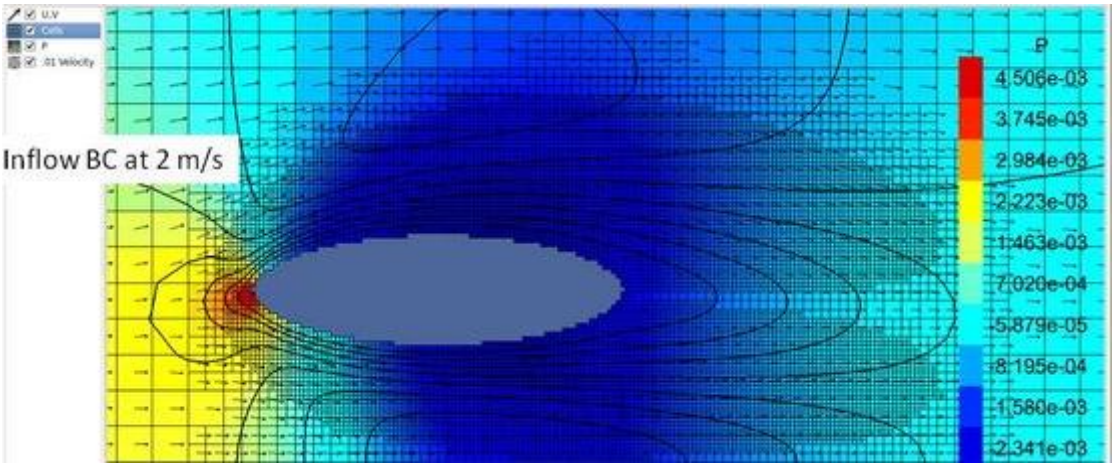


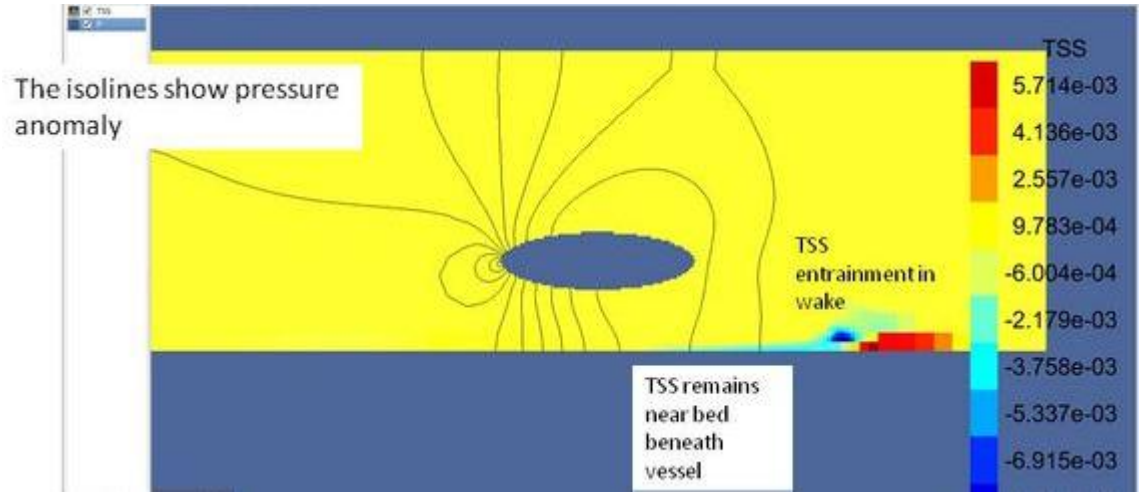
Figure 10.18. Pressure (contours) and flow vectors from Gerris for a 20 m hull 3 m above the seabed. Minimum cell is 5 cm. The max pressure anomaly at bow is 0.0004.

A second example (Figure 10.19) simulates the flow around the same hull moving at 2 $m \cdot s^{-1}$ and 6 m above the seabed. The highest resolution used was 23 cm. This case is somewhat different from the previous example. The largest pressure anomaly (P_{max} = 0.0045) is well above the bed, where the anomaly is ~0.0007 near the bow. A large region of negative anomalies occurs below the hull, with P_{min} = 0.0023. The isolines indicate the flow field. There is a large gradient above the bed and a region of uniform flow approximately equal in size to the hull, which coincides with the largest negative pressure anomalies.

Figure 10.19. Computation from Gerris for a 20 m hull 6 m above the seabed.



A. GFS computations of pressure and currents for 2 m/s and 6 mab. The AMR is displayed as rectangles with min cell of 23 cm. The bow wave pressure anomaly reaches 0.002 and the current isopleths indicate reduced flow beneath the hull and acceleration behind.



B. Total Suspended Solids field (contoured) and pressure anomaly isopleths for hull at 6 mab and speed = 2 m/s.

Total suspended solids (TSS) were implemented into the simulation using a GfsFunction from the input file:

```
Source TSS { return P > PMAX || P < PMIN ? 1.0 : 0.0; }
```

where the pressure anomaly is used to entrain TSS using: PMAX and PMIN = 10^{-6} and -10^{-6}, respectively. Figure 9B shows isolines of the pressure anomaly and the nondimensional value of TSS. TSS remains low beneath the hull because of the reduced flow and lack of vertical mixing. However, it has been entrained by the pressure anomaly at the forward part of the hull. It is finally lifted above the bed by turbulence and advection in the hull wake. The negative values indicate areas from which TSS has been transported and positive values are local areas of increased concentration.

These results are relevant to the 6.1 project ('Transport and Mixing of Terrigeneous Sediment in the Coastal Ocean')that is primarily developing Gerris as a littoral modeling system. The requirements for the NOVAS work are: (1) turbulence model; (2) sediment entrainment; and (3) the properties of the bed material. There is a small but growing literature on the impact of boat wakes on resuspension and water quality (e.g., Houser 2011; Donnelly and Walters 2008). They are more often studied for detection by remote sensing methods (e.g., Bunkin et al. 2011).

References Cited

Bunkin, A.F., Klinkov, V.K., Lukyanchenco, V.A., and Pershin, S.M., Ship wake detection by Raman lidar. *Applied Optics*, 50 (4), A86-A89, DOI: 10.1364/AO.50.000A86, 2011.

Donnelly, M.J. and Walters, L.J., Water and boating activity as dispersal vectors for Schinus terebinthifolius (Brazilian pepper) seeds in freshwater and estuarine habitats. *Estuaries and Coasts*, 31 (5), 960-968, DOI: 10.1007/s12237-008-9092-1, 2008.

Houser, C., Sediment Resuspension by Vessel-Generated Waves along the Savannah River, Georgia. *J. Waterway, Port, Coastal, and Ocean Eng.*, 137 (5), 246-257, DOI:10.1061/(ASCE)WW.1943-5460.0000088, 2011

Latz, M., Deane, G., Stokes D., and Hyman, M.. Developing a Predicitive Capability for bioluminescence signatures, ONR annual report, Grant Number N00014-09-1-0495, 2010.

Lee, S., E. Jin, H. Lee, and Isope. 2005. Evaluation of vertical plane dynamic stability by CFD, p. 168-172. *Proceedings of the Fifteenth. International Offshore and Polar Engineering Conference Proceedings*. International Society Offshore& Polar Engineers.

Lee, S. W., Y. S. Hwang, M. C. Ryu, I. H. Kim, and M. S. Sin. 2003. A development of 3000-ton class submarine and the study on its hydrodynamic performances, p. 363-368. In J. S. Chung, J. Wardenier, R. M. W. Frederking and W. Koterayama [eds.], *Proceedings of the Thirteenth. International Offshore and Polar Engineering Conference Proceedings*. International Society Offshore& Polar Engineers.

Popinet, S. 2003. Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries. *Journal of Computational Physics* 190: 572-600.

Popinet, S. 2009. An accurate adaptive solver for surface-tension-driven interfacial flows. *Journal of Computational Physics* 228: 5838-5866.

Popinet, S., M. Smith, and C. Stevens. 2004. Experimental and numerical study of the turbulence characteristics of airflow around a research vessel. *J. Atmos. Ocean. Technol.* 21: 1575-1589.

Rickard, G., J. O'callaghan, and S. Popinet. 2009. Numerical simulations of internal solitary waves interacting with uniform slopes using an adaptive model. *Ocean Modelling* 30: 16-28.

Tang, H., and T. Keen. 2011. Hybrid model approaches to predict multi-scale and multi-physics coastal hydrodynamic and sediment transport processes. *Sediment Transport.* Intech Open-Access.

Wu, J. M., Z. Y. Li, and Isope. 2005. Computational fluid dynamics analysis of an underwater towed system, p. 325-330. *Proceedings of the Fifteenth. International Offshore and Polar Engineering Conference Proceedings.* International Society Offshore& Polar Engineers.

Gerris Ice Dynamics

Introduction

This section discusses of ice modeling using Gerris. Preliminary test have been completed in the following directory: /home/keen/PROJECTS/ICE

Method

The idea behind these simulations is that the behavior of solids can be treated using a Newtonian fluid model with high viscosity. This has been demonstrated for a column of grains by Popinet, and fluid mud by Knoch and Malcherek (2011).

A sample simulation file is:

```
Define MAXSECS 60.0
Define IPRINT 1
Define TPRINT 60.
Define TSPRINT 60.0
Define Lref 100.0
Define Uref 1.0
Define RHOF -0.1
Define SMAX 1.0
Define RHO(Ice) (1000. * (1.0 + (Ice*RHOF/SMAX)))
Define GRAV -9.81

                0 GfsSimulation GfsBox GfsGEdge {} {
Time { end = MAXSECS }
VariableTracer {} Ice
PhysicalParams { L = Lref  alpha = 1./RHO(Ice) }
Refine 6
Init {} { U = 0 }
# AdaptVorticity { istep = 1 } { maxlevel = (x > 70.5 ? 0 : 6)  cmax = 1e-2 }
AdaptVorticity { istep = 1 } { maxlevel = 8  cmax = 1e-2 }
AdaptGradient { istep = 1 } { maxlevel = 8 cmax = 1e-2 } Ice
```

```
Init {} { Ice = { return (y < 30 || y > 35) ? 0.0 : 1.0; } }
#SourceDiffusion {} Ice 0.000001
SourceViscosity {} { return (Ice < 1.) ? 0.001 : 1.0; }

OutputTime { istep = IPRINT } stderr
OutputPPM { istep = IPRINT } ice.ppm  { min = 0 max = 1 v = Ice }
OutputGRD { step = TPRINT } u-%g.asc { v = U }
OutputGRD { step = TPRINT } w-%g.asc { v = V }
OutputGRD { step = TPRINT } s-%g.asc { v = Ice }
OutputGRD { step = TPRINT } l-%g.asc { v = Level }
OutputTiming { start = end } stderr
OutputSimulation { step = 10 } ice-%g.gfs
GfsOutputLocation   { step = TSPRINT } timeseries.dat 35.0 -1.25 0
}

GfsBox {
left = Boundary {
BcDirichlet U { return (y > 30 && y < 35) ? Uref : 0.0;  }
}
}
```

Results

A couple of simple ice representations have been completed. The results represent them. The images displayed in this report were made with the following graphics software: /common/gerris/devel/bin/gfsview2D *.gfs.

Compression Example

The mesh is initialized to a refinement of $2^6 = 64$ (Figure 10.20). The resulting background resolution is 100/64 or 1.56 m. A maximum refinement of $2^8 = 256$ is used as a function of ice concentration, resulting in the finest cells seen in the figures being 39 cm.

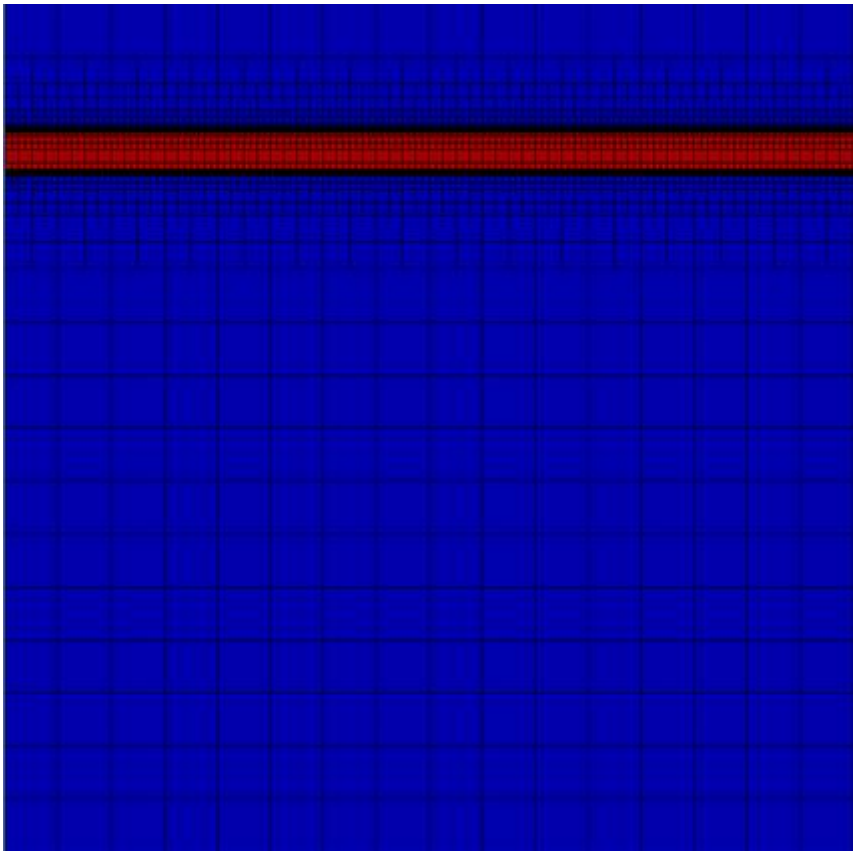
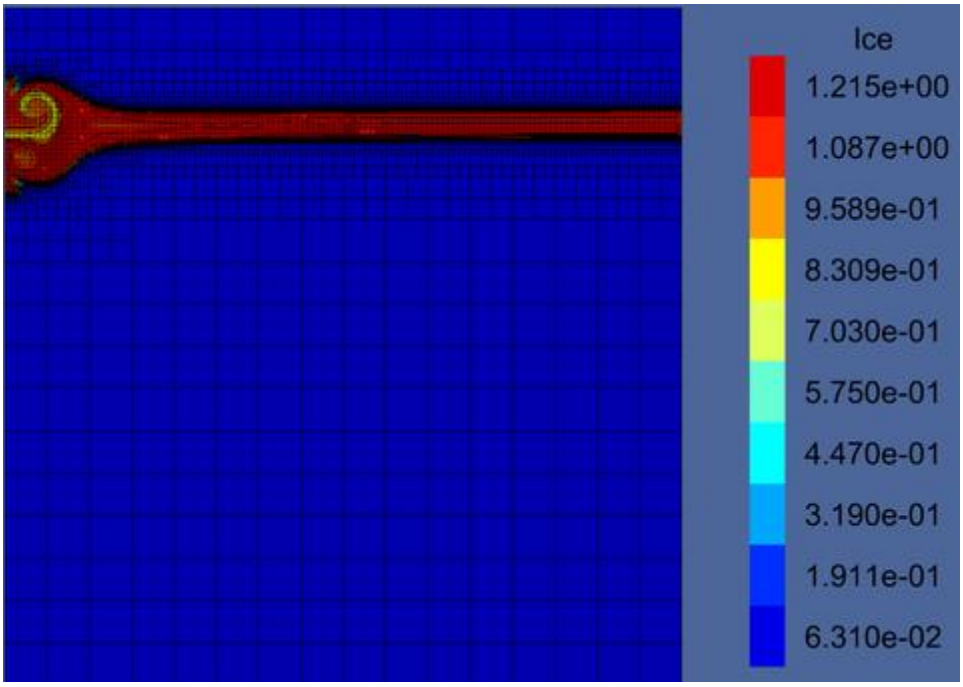


Figure 10.20. Initial condition for ice experiment. The red is solid ice.

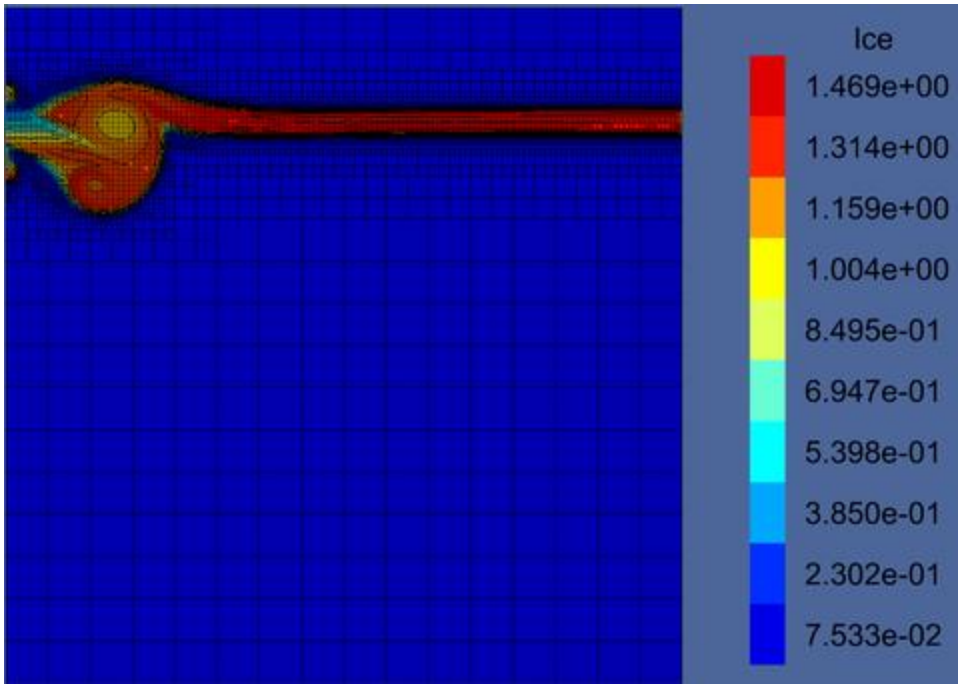
This simulation represents one side of a symmetrical compression zone in an ice sheet that is 5 m thick. The water depth is 100 m. The top of the ice sheet is also water. The left boundary is being compressed in the ice sheet only at 1 m/s. The right boundary is closed, representing an infinitely wide ice sheet. The entire simulation is 60 seconds.

As the ice is compressed, it bulges symmetrically because the fluid has the same density above and below it (Figure 10.21A). As the compression continues, it begins to flow and the density changes (Figure 10.21B). This simulation did not limit the density. Low values imply mixing with *water*, which represents void spaces and thus a lower bulk density. The large concentrations are unrealistic. This could be avoided by a cap on density, which should cause more deformation. The rate of compression is very unrealistic as well.

Figure 10.21. Ice properties calculated by Gerris. The density of the ice has changed due to the compression.



A. Time = 30 s.



B. Time = 60 s.

References

Denise Knoch and Andreas Malcherek (2011). A numerical model for simulation of fluid mud with different rheological behaviors. *Ocean Dynamics Theoretical, Computational and Observational Oceanography*, 61 (2-3), 245-256, 10.1007/s10236-010-0327-x.

Other applications

- Tamar River (On-line only)
- Mississippi Bight Tides (On-line only)
- Coupled Hydrodynamics and morphology (On-line only)
- Yellow-Bohai-East China Seas (On-line only)
- Wave Models (On-line only)
- Error Covariance in a Reduced Model (On-line only)

Appendices

Appendix A. Model Structure and Operation

Introduction

Gerris is a hybrid computer code. It is written in ANSI c to mimic object oriented programming. Specifically, Popinet (the author of GTS and Gerris) has implemented classes using c structures. He has gone to great lengths to emulate inheritance using a combination of pointers to structures and functions, and macros (c preprocessor directives). These details are useful for anyone wishing to modify or implement new modules. They are also helpful in understanding the potential use of the GfsFunction class.

Gerris is a modular software system that consists of four main components: (1) the Gnu Triangulated Surface Library (GTS); (2) the Gerris Flow Solver (GFS); (3) the GfsView visualization utility; and (4) Gnu Library (Glib). The GTS library is fundamental to the object oriented approach to its structure. Both GTS and Gerris were developed by Stefane Popinet using pre-existing libraries like those from the GTK+ project.

This section presents examples of the class structure of the GFS library, which is built on the GTS libraries in turn. The entire system of libraries is written using the c programming language in a manner that emulates object oriented programming as implemented in C++. This style is referred to as object-oriented programming in c.

The method of solving the Navier-Stokes equations for a given domain rests on generating a Cartesian grid that is intersected by these surfaces in order to conserve mass. These triangulated surfaces represent boundaries within the model domain. This approach grew from the need to accurately represent the interface between fluids/gasses with very different densities and viscosities (Popinet and Zaleski, 1999).

The second basic construction used in the GTS/Gerris system is the Quad/Octree disctretization. The implementation of these two concepts in solving the incompressible Euler equations is described by Popinet (2003). These requirements, the use of surfaces and time-varying grid adaptation, are basic reasons for the unique structure of the Gerris code.

The GfsSimulation Class

The fundamental class for running Gerris is a GfsSimulation, which is the structure containing all of the conditions associated with a specific simulation. This class inherits all of the attributes and functions of its parents.

```
struct _GfsSimulationClass {
  GfsDomainClass parent_class;
  void    (* run) (GfsSimulation *);
  gdouble (* cfl) (GfsSimulation *);
};
```

```
struct _GfsSimulation {
  GfsDomain            parent;
  GfsTime              time;
  GfsPhysicalParams    physical_params;
  GfsMultilevelParams  projection_params;
  GfsMultilevelParams  approx_projection_params;
  GfsAdvectionParams   advection_params;
  GtsSListContainer  * refines;
  GtsSListContainer  * adapts;
  GfsAdaptStats        adapts_stats;
  GtsSListContainer  * events, * maps;
  GSList             * modules, * globals, * preloaded_modules;
  GtsSListContainer  * solids;
  guint                thin;
  gboolean             output_solid;
  gboolean             deferred_compilation;
  gdouble              tnext;
  GfsVariable        * u0[FTT_DIMENSION];
  GHashTable         * function_cache;
};
```

Most of the classes listed in this structure are found in the simulation file; Domain, Time, PhysicalParams, MultilevelParams, AdvectionParams, and AdaptStats. The most basic of these is the GfsDomain, which implements the physical region of the earth to be simulated through the GtsWGraph class. As an example of the modularity of the code, we list in pseudocode the algorithm for reading the domain data:

- main<--gfs_simulation_read<--gfs_domain_read<--gts_graph_read<--(* klass->read) **aka** *graph_read*

where "<--" indicates a called function to the right. The simulation and domain data are read by GFS functions whereas the graph data are read by a GTS function. Function, *gfs_simulation_read* calls the (* klass->read) member of a GfsSimulationClass, which is *simulation_read*. A similar procedure is applied to the GfsDomainClass and GtsGraphClass.

The GtsSListContainer members of the GfsSimulation structure store the necessary parameters for a simulation. For example, a print statement was inserted in function *container_add* to identify how frequently container (pointers) were written to. This list of pointers was correlated to labels written in other functions to produce the following estimated container contents:

- ...592(1)-- MapProjection

- ...528(2)-- Refine and RefineSurface (bath.gts)
- ...720(19)--Init (**AM2.gts** and **BM2.gts**), Solid (**bath.gts**), SourceCoriolis, Init (U and V), EventHarmonic (3), EventStop, OutputTime, OutputProjectionStats, OutputSimulation(2), OutputPPM (2), OutputGRD (4), EventScript
- ...464(1)-- Solid (**bath.gts**)
- ...400(1)-- SourceCoriolis
- ...560(1)-- SourceCoriolis
- ...928(1)-- SourceCoriolis
- ...072(1)-- Unknown
- ...296(1)-- Same Unknown

These are the last 3 digits from the pointers to the GtsContainers, followed by a brief description of items I think are added to them. The numbers in parentheses are the number of items in each container. For example, pointer number *464 is going to be **solids** whereas *720 will be **events**.

The GfsDomain Class

The GfsDomain is part of the GFS library but it accesses the GTS software as well as the Glib functions. Its main member, however, is the *GtsWGraph* (i.e., GtsGraph parent):

```
 struct _GfsDomain {
   GtsWGraph parent;
   int pid;
   GfsClock * timer;
   GHashTable * timers;
   GtsRange timestep;
   GtsRange size;
   gboolean profile_bc;
   GtsRange mpi_messages;
   GtsRange mpi_wait;
   guint rootlevel;
   FttVector refpos;
   FttVector lambda;
   GArray * allocated;
   GSList * variables;
   GSList * derived_variables;
   GfsVariable * velocity[FTT_DIMENSION];
   GSList * variables_io;
   gboolean binary;
   gint max_depth_write;
   FttCellInitFunc cell_init;
   gpointer cell_init_data;
   gint version;
   gpointer array;
   gboolean overlap; /* whether to overlap MPI communications with
computation */

   /* coordinate metrics */
   gpointer metric_data;
```

```
   gdouble (* face_metric)       (const GfsDomain *, const FttCellFace *);
   gdouble (* cell_metric)       (const GfsDomain *, const FttCell *);
   gdouble (* solid_metric)      (const GfsDomain *, const FttCell *);
   gdouble  (*  scale_metric)          (const  GfsDomain  *,  const  FttCell  *,
FttComponent);
   gdouble  (*  face_scale_metric)  (const  GfsDomain  *,  const  FttCellFace  *,
FttComponent);

   /* Object hash table for (optional) object IDs */
   GHashTable * objects;

   /* total number of parallel processes */
   int np;

   /* real time */
   GTimer * clock;
   GPtrArray * sorted; /**< array of sorted boxes */
   gboolean dirty;     /**< whether the sorted array needs updating */
 };
```

The GfsDomain structure contains many of the required classes (structures) of the problem domain. The first member of the GfsDomain structure is a (GtsWGraph parent), which introduces the data read from the file **bath.gts**. I have verified that the (GtsGraph *) returned by *gts_graph_read* is present in *gfs_domain_read* as &(domain->parent.graph).

Function *gfs_domain_class* is called by GFS_DOMAIN and it returns a GfsDomainClass object, which is cast as a GfsDomain. Functions *simulation_read*, *domain_read*,, and *graph_read* are not present by name, but as the "read" members of their respective GtsObjectClassInfo structures through inheritance.

The return value of *gts_graph_read* is checked to be a GfsDomain by a call of the macro, GFS_DOMAIN. This has the usual behavior of instantiating any required classes and casting the GtsGraph to a GfsDomain. This casting is a more complex because there is an intermediary structure between the GtsGraph and the desired GfsDomain (GtsWGraph).

The GfsInit Class

The GfsInit is an initialization event and thus it uses functions from the GfsEventClass. Specifically, it is a GfsGenericInit, which is *exactly* a GfsEvent. The GfsEvent class functions are contained in file, **event.c**. The "GtsObjectClassInitFunc" for the GfsEventClass is *gfs_event_class_init*. This class is initialized by a call to *gfs_event_class* by function *gfs_classes* at the beginning of a simulation.

```
typedef struct _GfsInit          GfsInit;
struct _GfsInit {
  /*< private >*/
  GfsGenericInit parent;
  GSList * f;
};

typedef struct _GfsEvent         GfsGenericInit;
```

```
typedef struct _GfsEventClass    GfsGenericInitClass;

typedef struct _GfsEvent         GfsEvent;
typedef struct _GfsEventClass    GfsEventClass;
struct _GfsEvent {
  GtsSListContainee parent;
  gdouble t, start, end, step;
  guint i, istart, iend, istep;
  guint n;
  gboolean end_event, realised, redo;
  gchar * name;
};
```

A GfsInit structure contains a GSList pointer (* *f*). This pseudoclass is initialized by *gfs_init* using function *gfs_init_class*(). The "read" function is initialized in *gfs_init_class_init* by *gts_object_class_new* (in the usual manner) to be *gfs_init_read*. I checked the pointer value for *gfs_init_read* and it is the same function called on ~line 362 of *simulation_read* (file simulation.c). A print statement verifies this. An event is a GtsSListContaineeClass. Function *gts_object_class_new* is called by *gfs_event_class* to construct a GtsSListContaineeClass. This function calls *gts_object_class_new* with the "gfs_event_info" structure containing the name (actually pointer) of the "class_init_func" (*gfs_event_class_init*).

GfsSurfaceClass

The triangulated surface is a basic construction implemented in Gerris to represent 2D fields. These surfaces are used for bathymetry and boundary conditions like water surfaces computed from tidal constituents. The GfsSurface structure contains a GtsSurface member, and thus inherits face, edge, and vertex members from it as well as including independent position information.

```
typedef struct _GfsSurface    GfsSurface;

struct _GfsSurface {
  /*< private >*/
  GtsObject parent;
  GtsVector rotate, scale, translate;
  gboolean flip;
  GfsFunction * f;
  GtsMatrix * m;
  GNode * bbtree;

  /*< public >*/
  GtsSurface * s;
  gboolean twod;

  GtsFaceClass * face_class;
  GtsEdgeClass * edge_class;
  GtsVertexClass * vertex_class;
};
```

There is no difference in the descriptions for these surfaces; the parent class of a GfsSurface is a GtsSurfaceClass by inheritance through its (GtsSurface *) member. These surfaces are read by the same *read* function, which is defined through the GtsObject parent. This section will discuss the incorporation of the *GfsSurface* only. There is no GfsSurfaceClass structure because the GtsSurfaceClass structure, which complements the GtsSurface structure in the pseudo-object-oriented programming style used in GFS and GTS, is inherited from the (GtsSurface *s) member (see above):

```
struct _GtsSurfaceClass {
  GtsObjectClass parent_class;
  void (* add_face)    (GtsSurface *, GtsFace *);
  void (* remove_face) (GtsSurface *, GtsFace *);
};
```

Reading a GfsSurface from a File

A GfsSurface is actually a subclass of the GtsSurface class. The surface is read entirely using the inherited GtsSurface class functionality. The reading functionality has been tested for tidal elevation data for the Mississippi Bight. Two files are read for a typical tidal simulation: **AM2.gts** and **BM2.gts**. These files are opened in *read_surface* (GFS/utils.c). The **bath.gts** file is opened by *surface_read* (GFS/surface.c) and cast as a GtsFile before being passed to *gts_surface_read*.

The function *surface_read* is assigned to the "read" member of the parent GtsObject class. There is no explicit call of *surface_read*. It is invoked implicitly in a call to a function like *simulation_read*. Function *surface_read* (surface.c) instantiates a (GfsSurface *surface) and its (GtsSurface s) member using *gts_surface_new*. The (GfsSurface *surface), to which the bathymetry surface is assigned, is initialized by the macro GFS_SURFACE as a GtsObject. This function assigns/initializes face_class, edge_class, and vertex_class members of the GtsSurface. Function *gts_surface_read* then assigns values to surface->s. The *x, y*, and *z* values from the file must be processed into the vertices, edges, and faces of a surface. The specific method is not the same for all three member classes.

The *add_face* function (see the GtsSurfaceClass definition above) is defined in file, graphic.c (GFS). It appears to be a wrapper for *gts_surface_add_face*, which it calls with the edges that have been passed. If all of the faces are read successfully from the **gts** file and inserted into the hashtable and GSList, function *gts_surface_read* returns 0.

The functions used to read the gts files are a good example of using the available libraries, supplemented by user-defined functions:

```
read_surface                 GFS function
  gts_surface_read           GTS function
    gts_point_read           GTS function
      gts_surface_add_face   GTS function
        g_hash_table_insert  Glib function
```

The primary difference between this GfsSurface that is produced from the **gts** file and the GtsSurface is the inclusion of parameters for transforming the surface.

Transforming a GfsSurface

The GfsSurface class extends the GtsSurface class by including parameters for transforming the surface:

```
rotate[ ]: rx = rotation around x axis
           ry = rotation around y axis
           rz = rotation around z axis
scale[ ]:  sx = scaling along x axis
           sy = scaling along y axis
           sz = scaling along z axis
translate[ ]: tx = translation along x axis
           ty = translation along y axis
           tz = translation along z axis
scale = uniform scaling
flip = flip axes
twod = 1 for a 2D file
implicit = NULL
```

A number of transformations are completed based on these parameters. The user can supply functions to complete some of them. The following values are assigned in *gfs_surface_init* for a GfsSurface:

```
scale[0], [1], [2] = 1
flip = FALSE (0)
```

After a GtsSurface *s has been defined by function *gts_surface_read* and the GfsFileVariable has been initialized in function *surface_read*, the surface is processed using *gts_file_assign_variables* (file GTS/misc.c).

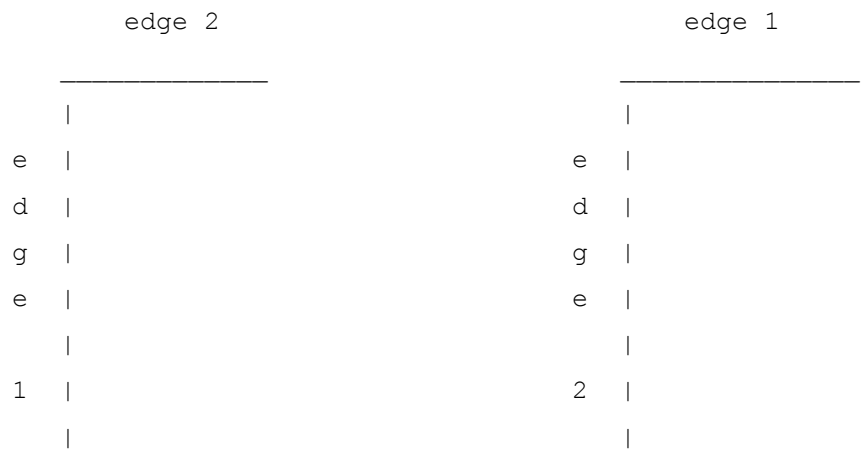
Some of the surface transformation parameters are initialized in *gfs_surface_init* as members of a GtsSurface, and **INDEPENDENTLY** as members of the GtsFileVariable structure array var[]. The var[] structure includes *type, name, unique, data, set, line*, and *pos* members. When *surface_read* initializes var[], it only assigns the first 4 elements for each transformation vector listed above. Furthermore, scalars *scale* and *implicit* are assigned local variables (*scale* = 1 and *implicit* = FALSE = 0). This leaves the *set, line,* and *pos* member/elements unassigned. Overall, 14 rows are assigned to var[]. The last is GTS_NONE to signal the end of processing.

Function *gfs_assign_variables* calls *gts_file_assign_start* first. This function marks a temporary (GtsFileVariable *) as unassigned (*set* = FALSE = 0) for all of the variables passed from *surface_read*. The result is to assign the var[].set member to FALSE (0) for each element of var.

Function *gts_file_assign_next* is then called by *gts_file_assign_variables*. This function checks that each var[] (i.e., *rotate*[3], *scale*[3], *translate*[3], *flip*, and *twod*) was not previously assigned and that it is the correct type. It then returns the GtsFileVariable with the values from the

simulation file (e.g., **tides.gfs**). The transform values for flip are not set when it is assigned in the GfsInit block that reads the **BM2.gts** and **AM2.gts** files.

Function *gfs_surface_transformation* is then called with the GfsSurface *, *rotate, translate, scale, flip*, and (GtsMatrix *) parameters. If *flip* is true (a value of 1), the function *gts_surface_foreach_face* is called with the *gts_triangle_revert* argument passed. This does not occur for the realistic example of flip = 0. The function *gts_triangle_revert* changes the orientation of a triangle, turning it inside out. For example, for a given face



The edges are not necessarily oriented exactly N-S and E-W; they do, however, appear to be trending those directions. The actual class modified in gts_triangle_revert is:

```
 (GtsTriangle  *t)->(GtsEdge  *e1)->(GtsSegment  segment).(GtsVertex  *v1)-
>(GtsPoint p).(gdouble x)
```

where *t* is passed from *gts_surface_foreach_face*. This is accomplished by swapping the order of the edges in the hash table, *faces*, which is a member of the GtsSurface structure. A pointer to the function *gts_triangle_revert* is passed to the *g_hash_table_foreach* function; edge 2 becomes edge 1, and edge 1 becomes edge 2 while edge 3 is unchanged. It looks like edge 1 is oriented ~N-S and edge 2 is ~E-W. It doesn't appear that the flip transformation will change the latitude. Note that the GSList associated with the GtsFace that contains the GtsTriangle is NOT updated to reflect this flipping. However, this action will change the hash table values for the faces.

Bathymetry as a GfsSolid

The bathymetry is transformed/recast as a GfsSolid object. The Solid object class has a new GtsObject created by *gts_object_new*, which calls *gfs_solid_init*. The file is read by (*klass->*read*) on line 353 of file, simulation.c. The pointer for this "read" function is the same as assigned in *gfs_surface_class_init*. The name of this object as printed by *surface_read* is "GfsSurface". It has the same pointer value as the Solid class from *gfs_solid_class*. The "read" function pointer printed by *gfs_solid_read* (the "read" function for the solid class) is different from that assigned in 'gfs_solid_class_init.

```
typedef struct _GfsSolid         GfsSolid;

struct _GfsSolid {
  GfsEvent parent;
  GfsGenericSurface * s;
};
```

A GfsGenericSurface is equivalent (typedef statement) to a GtsObject. A GfsEvent contains a GtsSListContainee member as well as scalars for timing and tracking of events. The *read* function is the same as assigned in *gfs_surface_class_init*. This object is a *GfsSurface* with the same pointer value as the GfsSolid class.

The bathymetry for a simulation is stored in a container. This takes us to the GtsHashContainer class contained in the GtsGraph structure. The bathymetry vertices will be contained in a GSList directly in the GfsSimulation object:

1) (GfsSimulation *sim)->(GtsSListContainer *solids)->(GSList *items)->(gpointer data)

or

2) (GfsSimulation *sim)->(GtsSListContainer *solids)->(GtsContainer c).(GtsSListContainee object).(GSList * containers)->(gpointer *data)

Tidal Constituents as Gfsinit (GfsEvent) Objects

Tidal data are different from the bathymetry because they are not static. Something must be done with the constituents every time step. This necessitates their processing as events.

Reading the GtsSurface Data

The **AM2.gts** and **BM2.gts** files are processed by function, *read_surface*. The surface data are temporarily read into the hash table member of a GtsSurface with no additional characteristics. The function, *gts_surface_add_face*, is called by *gts_surface_read* and passed the input arg, (GtsSurface *surface) as well as a (GtsFace * new_face). The Glib function, *g_hash_table_insert*, inserts the current face from the file in the GHashTable *faces member of a GtsSurface with the face values used as the key. A print statement in *gts_surface_add*_face prints the data in the new face read from the **AM2.gts** and **BM2.gts** files exactly as they appear in the files.

Now that a set of faces have been read from a file, they must be incorporated into the Gerris framework. Function *read_surface* (utils.c) returns a GtsSurface but *read_surface* is called by *function_read*, which assigns the structure pointer to the (GtsSurface *s) member of a GfsFunction. This can be diagrammed like this:

```
GfsFunction.                          f
  ...
  GtsSurface                          s
    GtsObject                         object
    GHashTable  *                     faces
    ...
```

```
     GtsVertexClass                    vertex_class
       GtsPointClass                   parent_class
         GtsObjectClass                parent_class
         gboolean                      binary
       void   (*intersection_attributes) (GtsVertex *, GtsObject *, GtsObject
*)
   ...
```

To recap, the point data are placed in the (GtsSurface *s) that is passed as a pointer in the GfsFunction structure. This allows a range of processing to occur for the GtsSurface data that are to be applied as a GfsInit class, which is a GfsEvent. This structure contains a GtsSListContainee structure, which is the storage location for the tidal constituents. The instructions for applying the data are included through the GfsFunction mapped above.

Processing Boundary Conditions as *GfsInit* Objects

This section is referring specifically to the boundary condition of the tidal amplitudes read from the files, **AM2.gts** and **BM2.gts**. The GfsInit is a special kind of event class that is only executed once. The processing is supplied by the user in the gfs library. This is a little awkward but it works.

After the tidal constituents are read from the ***.gts** files, the classes containing them are added to an event (GfsInit) container using *gts_container_add*. The (GfsSimulation->(GtsSListContainer *events)) structure is the container for the current object (GfsInit). Function *gts_container_add* uses the "klass->add" function to add the GfsInit structure pointer. Since (*events) is a GtsSListContainer, the "init" function is probably going to be *slist_container_class_init*, which initializes the "add" function to be *slist_container_add*. In fact, a print statement verified that GTS/*slist_container_add* is the "add" function, but it calls the "add" function for its parent, which was verified to be a GtsContainer. The "add" function for a GtsContainer is *container_add*. This function (GTS/c*ontainer_add*) calls its "add_container" function. The "add_container" member of a GtsContaineeClass is assigned NULL in function, *containee_class_init*. However, a GtsSListContaineeClass structure is merely a wrapper for a GtsContaineeClass (its parent_class). Furthermore, a GtsSListContaineeClass "add_container" member is initialized to *slist_containee_add_container* in function, *slist_containee_class_init*. This is verified by a print statement in *slist_containee_add_container*. Finally, the object passed from *simulation_read* (GfsInit) has the same pointer value in all of the intermediate functions:

simulation_read passes (GfsSimulation *sim, GfsInit *object) to:
  gts_container_add passes (GfsSimulation *sim, GfsInit *object) to:
    slist_container_add as (GtsContainer * c, GtsContainee * item), which passes
                (GtsContainer *c)->items, GtsContainee *item) to:
      g_slist_prepend to update the singly linked list of simulatio objects.

It then passes (GtsContainer *c, GtsContainee *item) to:
      container_add as (GtsContainer * c, GtsContainee * item), which passes
              (GtsContainee * item, GtsContainer * c,) to:

slist_containee_add_container as (GtsContainee * i, GtsContainer * c),

which prepends the GtsContainer *c to the singly linked list

(GtsSListContainer *events) associated with the simulation.

We want to know where the GfsInit event is stored so that it can be retrieved at will. We would like to print out these data in simulation_read. First, how do we get the keys to (GtsSurface *)->(GtsFace *)? A GfsInit object has no associated class. Instead, the macro GFS_IS_EVENT is used to see if it is an event.

```
#define GFS_IS_EVENT(obj)   \
        (gts_object_is_from_class (obj, gfs_event_class ()))
```

is defined in event.h. Function *gfs_event_class* returns a GfsEventClass pointer that is common to all such classes. The *gts_object_is_from_class* function is a "static inline gpointer" function in the header file, gts.h. It is passed a gpointer object and a gpointer klass. The passed object is cast to a (GtsObject *), which has a (GtsObjectClass *klass) member. The passed class pointer (klass) is compared to the hierarchy of parent classes for the recast object. When a match is found, its value is returned.

The GTS library functions are used to place the GfsInit object within its proper location in the GfsSimulation structure. The (GfsSimulation *) is cast to a (GtsContainer *) when it is passed to *gts_container_add*. The (GfsInit *) object is cast to a (GtsContainee *). The purpose here is to make use of existing library functions to store data that would otherwise be placed in arrays or a number of different variables. The GfsInit structure is a good example of how difficult this can be. For example, It has only two members, a (GfsGenericInit parent) and a (GSList *f). The GfsGenericInit is a synonym for GfsEvent (I don't know why they bothered but maybe it was legacy). The parent of a GfsEvent is a GtsSListContainee. The "pattern" for related structures like GtsSListContainee and GtsContainee is that the GtsContainee is included in the GtsSListContainee, which also includes a (GSList *) for the containers contained within it. Thus, when we cast a (GfsInit *) as a (GtsContainee *), we are referring to only the (GtsObject object) member of the GtsSListContainee structure. It is of note that the GtsContainee structure has only a GtsObject member and nothing else. In other words, a GfsEvent is simply a GtsObject with some ordering information contained in a singly linked list. A GtsObject only contains information about the class and no data. Of course, a GfsEvent also contains some variables that control time dependency for the event.

The input arguments to *gts_container_new* are passed without modification to the "add" function member of the GtsSListContainerClass structure. The "add" member is not part of the GtsObjectClass, but it is introduced by the initialization functions that are part of a GtsObjectClass structure. The basic container initialization is all part of the GTS library. This approach allows the user to implement new classes like the GfsEventClass. The "add" function is assigned in the class initialization functions for most classes, following the GTS prototype.

Function *gts_container_add* calls *slist_container_add*. The arguments passed to *slist_container_add* are unchanged (GtsContainer * c, GtsContainee * item). The item is

prepended to the (GSList *items) member of the GtsSListContainer structure. The "item" is the pointer to the GfsInit structure into which we wish to place the tidal constituents. This can be represented by (GfsSimulation *sim)->(GtsSListContainer *events)->(GSList *items). The other member of the (GtsSListContainer *events) object is the (GtsContainer c). Note that "c" is not a pointer.

Following the GTS prototype, the "add" function for the parent class of the GtsSListContainerClass (i.e., GtsContainerClass) is called with the input arguments passed unaltered. This function is called, *container_add*. The input arguments to *container_add* are not implicitly recast in the argument list. This function is part of the GTS library but it's purpose is ambiguous. It consists of a call to the "add_container" function of the GtsContaineeClass. This structure has a member as follows:

```
Void     (* add_container)    (GtsContainee *, GtsContainer *);
```

The GtsContaineeClass "init" function (*containee_class_init*) assigns NULL to the "add_container" member of its structure. However, the GtsSListContaineeClass structure follows the procedure for SLists, and contains a (GtsContaineeClass parent_class) member. This means that when the "init" function for this class is called (i.e., *slist_containee_class_init*), the "add_container" member of a GtsContaineeClass is assigned a value of *slist_containee_add_container* because this "init" function includes a cast to a GtsContaineeClass using the macro, GTS_CONTAINEE_CLASS. However, the arguments for *slist_containee_add_container* must match those given in the GtsContaineeClass structure, namely (GtsContainee *, GtsContainer *), which is reversed from the input arguments to function, *container_add*.

The input (GtsContainee *i) is cast to (GtsSListContainee *item). Noting that the GtsSListContainee structure has only two members, (GtsContainee containee) and (GSList *containers), this cast populates the containee member of the GtsSListContainee object. The (GSList *containers) list is searched for the (GtsContainer *c) in the input (GtsContainee *i), which has been cast to (GtsSListContainee *item). This means it now has a GSList attached to it to keep track of the entries. The (GtsContainer *c) is placed at the beginning of this singly linked list (item->containers) using the Glib function, *g_slist_prepend*.

To review, the original (GtsObject *object) created in *simulation_read* contains the tidal constituents. Furthermore, the input (GtsContainee *i) variable points to a member of the simulation class that is created when the simulation file, **tides.gfs**, is read.

(GfsSimulation *sim)->(GtsSListContainer *events)->(GtsContainer c). ... (GfsFunction *f)

->(GtsSurface *s)->(GtsHashTable *faces)

Overview of a simulation

The primary input to Gerris is through the simulation file (*.gfs). An example will be referred to in this section, which will refer to program units as the file is read. The sample is **tides.gfs**. The tests referred to in this document include an original gerris example file and one I am attempting to run for the Mississippi Bight (MSB). The examples are in

TESTS/ GFS_TESTS-LOCAL_BUILD/Cook_Strait_Tides

GTS_FILE_INPUT

This report will walk through the file, **tides.gfs**, as it is read by the functions in the GFS/GTS libraries. I believe this is necessary after spending a lot of days following function and structure pointers with ambiguous names while attempting to locate an error in processing files for tidal constituents for MSB. The example is similar to the MSB case except that it is located in the southern hemisphere--latitudes are negative. This is the only difference that is readily seen.

Initialize the Gerris Simulation

The function *gfs_init* is called by *Gerris* to instantiate a GfsSimulation object (structure). This function is hard-wired to instantiate all of the Gerris classes to make sure they are available to create objects later: e.g., GfsOcean. These classes are initialized using the general GTS format: e.g., gfs_ocean_class, which will initialize the info member of the GtsObject structure i.e, "GfsOcean"). The initialization function (GtsObjectClassInitFunc) is assigned the location of the user-supplied function, like *gfs_ocean_class_init*. The function, *gts_object_class_new*, is then called with the name of the class function (e.g., *gfs_simulation_class*) passed to GTS_OBJECT_CLASS macro to initialize. This function (*gts_object_class_new*) creates a hash table for the classes and their parents.

The simulation file (**tides.gfs**) is opened in the main program, *gerris* (file **gerris.c**). It is passed on command line. The c function fopen is directly called from *Gerris* with the first argument in the command line as its name. This file pointer is used to generate a new GtsFile object with *gts_file_new*.

In order to use the many options that come with the GTS and Glib libraries, it is necessary to initialize a structure from the regular (ascii) file that was opened with fopen. This is done by the GTS function, *gts_file_new*. The GtsFile structure is located in file **gts.h**. The function *gts_file_new* is located in **misc.c**. The first act in *gts_file_new* is to call *file_new*, which initializes a GtsFile structure and returns a pointer to it. *Gts_file_new* then assigns the c file pointer to the file pointer that is a member of this structure. Several special character variables are assigned: type = "\0"; error = NULL; next_token = "\0"; delimiters = "\t"; comments = GTS_COMMENTS ("#!"); and tokens = "\n{}()=". It is passed the pointer to the GtsFile that includes the file, **tides.gfs**. The function members that indicate line and cursor position have been set to the start of the file.

A frequently used function is now called for the first time, *gts_file_next_token*. At this point, the input file is open to line 1, column 1. The cursor is advanced through the file ignoring any comment tokens (#). Control returns to *gerris* with the input file at this line.

Begin Reading the simulation file

The entire file is read by the following line:

```
if (!(simulation == gfs_simulation_read (fp))) {
  ...(error processing)
}
```

where *fp* is a pointer to the already-open simulation file. This section will describe what occurs when *gfs_simulation_read* is called by *gerris*.

The next line in the simulation file is

```
Define M2F (2.*M_PI/44700.)
```

This is a definition of the m2 tidal frequency. These are GFS macro definitions that are skipped by *gfs_simulation_read*, which moves down to read the number of GfsBoxes (nodes = 1) in the simulation. After the number of GfsBoxes has been read, function *gfs_domain_read* is called with the file pointer as its only argument. This function returns a GfsDomain pointer. The GfsDomain structure includes several kinds of parameters. The parent is a GtsWGraph,which is a weighted graph. A graph is a set of vertices connected by edges. A weighted graph associates a label (weight) with every edge in the graph. The weighted graph can be used to formulate the shortest path problem. The GtsWGraph structure contains a GtsGraph and a scalar, weight. The GtsGraph is a hash container with classes for the graph, its nodes, and its edges. What this means is that the domain is a weighted graph. The domain also includes timers,timestep parameters, mpi parameters,tree parameters, variable linked lists, the velocity array, cell initialization function pointers, grid metric arrays, and an object hash table.

Initialize the Domain Graph

The file pointer is now passed to *gts_graph_read*. This is a GTS function, which means that it will read a graph from a standard **gts** formatted file. No new characters have been read from the file, so the last number read, which was the number of GfsBoxes (referred to as nodes sometimes), is the number of nodes in the domain graph.

Initialize graph data (i.e., vertices, edges, and faces) and GfsOcean module. The next integer (0) of the simulation file is assigned to the number of edges of the graph. The module name, GfsOcean is read by *gts_file_next_token* on line 1409 of GTS/graph.c. Function *gts_graph_read* initializes "GtsGraph", "GtsNode", and "GtsEdge" classes. It calls *gts_object_class_from_name* to get a pointer to the appropriate class for the name from the file (GfsOcean). A GfsOcean structure contains a GfsSimulation and a GfsDomain structure. The GfsDomain contains a GtsWGraph and thus a valid pointer is generated for the input data.

A new GtsGraph object is created from the GfsOcean class that was initialized in *gfs_init*, and the pointer to the parent class is assigned to the graph_class member of the GtsGraph. The class *read* member is then called with the input file pointer and the new GtsGraph to receive the graph data. Function *gts_graph_read* calls the (*klass->*read*) function for the GfsOcean class (*ocean_read*). The *read* function (*ocean_read*) is passed a (GtsGraph *) after the function, *gts_file_next_token*, has been called to update (fp->token->str) to "GfsBox" instead of "GfsOcean". This is obvious from the klass pointer being accessed by the function *gts_object_class_from_name* before the name has been changed.

Initialize GfsOcean Class

Input the data for the GfsOcean class. The *ocean_read* function is passed a pointer to the GtsGraph (cast to a GtsObject **) and the GtsFile structure pointer. A GfsSimulation object is

created if necessary. The value of lambda.z is set to 1/maxlevels but this is noted as requiring change with a /* fixme */ note.

There is no GfsOceanClass. This is a pseudoclass (or ghost) that is not explicitly declared. Instead, the *gfs_ocean_class* function returns a GfsSimulationClass. Thus, when (*gfs_ocean_class*()) is cast to a GtsObjectClass by *ocean_read*, the parent is a GfsDomainClass (parent of a GfsSimulationClass). The initialized (klass->*read*) for this class is *domain_read* (file **domain.c**). Read the simulation file with *simulation_read*. The function, *domain_read*, is called by *ocean_read*.

Create a Domain/Graph Structure

Initialize the grid defined in the simulation file. Initialize (GtsFileVariable var[ ]). The "read" member of the (GfsDomainClass->parent) is called next with file, **tides.gfs** as an argument. This is the GtsWGraphClass, but it has no "read" member initialized in function, *wgraph_class_init*. Its parent is a GtsGraphClass, which has the "read" member assigned a value of *graph_read*.

Function *domain_read* calls function *graph_read* to create the GfsBoxes for the domain. This "read" member (*graph_read*) is a member of the GtsObjectClass; it is accessed through a sequence of parents (i.e., inheritance). This "read" occurs on line 222 of file domain.c in function, *domain_read*. In fact, a *printf* statement in GTS/*graph_read* prints next. Note that this is *graph_read* and not *gts_graph_read*. This function (*graph_read*) updates the class pointer from the current token, "GfsBox". It generates a GtsNodeClass and reads the next token, which is "GfsGEdge". A GfsBox structure includes the GtsGNode structure as a parent. The analogous class is the GfsBoxClass structure, which is a simple wrapper for a GtsGNodeClass. The GtsGNode includes a GtsSListContainer and a scalar, level. The GtsGNodeClass includes the GtsSListContainerClass and function pointers, "weight" and "write". A GtsGEdgeClass is created and the next token is read, "{", and the function returns to *domain_read*.

Assign Variables

Function *domain_read* calls *gts_file_assign_variables* to assign variables to the GtsFile structure. A lot happens in the next few lines of code. The (...*file_assign*...) functions are located in file, GTS/**misc.c**. This function is a while loop that calls *gts_file_assign_next* as long as there is a valid token in the input file. A token appears to be a string with no blanks.

Function *gts_file_assign_start* is called to initialize the "set" member of the GtsFileVariable structure. Then, function *gts_file_assign_next* reads the simulation file one token at a time and places the read values into the *var*[] array until a closing "}" is encountered.

...Control returns to *domain_read*
...Control returns to *simulation_read*

Process Keywords from Simulation File

There are two kinds of input tokens that are processed differently. First are the keywords hardwired into the *simulation_read* code:

- GfsDeferredCompilation
- GfsModule
- GfsTime
- GfsPhysicalParams
- GfsProjectionParams
- GfsApproxProjectionParams
- GfsAdvectionParams.

These keywords are associated with read/initialization functions that are hardwired in a series of if/else statements. The input data from these blocks are placed in the appropriate structures within the (GfsSimulation *sim) class, e.g., sim->*physical_params*, using the Glib function, *g_slist_prepend*. The GfsPhysicalParams class functions are contained in file, GFS/**simulation.c**.

General objects are read next in the following sequence for the tides.gfs file:

- MapProjection
- Refine
- Init
- Solid
- RefineSurface
- SourceCoriolis
- Init
- EventHarmonic objects
- EventStop
- OutputProjectionStats
- OutputSimulation objects
- OutputPPM
- OuputGRD objects
- EventScript

Apparently, the GfsOcean "module" is not actually a GModule. It is not processed in the same manner as the (GModule map) from the **tides.gfs** file. Function, *load_module*, is called to read and prepare a map. Call function, *gfs_time_read*, to read the GfsTime variables. Call function, *gfs_physical_params_read*, to read the simulation parameters. This function has several keyword options available in if/else blocks: g; L; and alpha. The keywords, "g" and "L" use a simple parsing function called, *gfs_read_constant*. The other keyword, "alpha", however, uses a *gfs_function_read*.

GtsObjects

Process GtsObject keywords that are user defined. It is not clear why the input changes from the keywords described above to more generic GtsObjects, but all of the other input tokens

from the simulation file are processed in the final "else" block. This may be because they are not a fundamental part of the Gerris engine, the CFD code. They thus have more complex functionality that has not been predefined as part of the fundamental model. Most of these functions, however, are part of the main library and are not included in the "modules" subdirectory. All of the possible GtsObjects that can be constructed are initialized in *gfs_classes*, which calls their constructors and returns pointers to their structures (GtsObjectClass *). New ones must be present in this function.

The first GtsObject in the simulation file is a GfsMapProjection. The files associated with the map projection are included in file, **map.c**. The user-defined GtsObjects are constructed from the names, so the appropriate initialization functions must be available. Function *gts_object_new* then allocates a pointer for the class and passes the class pointer to *gts_object_init*, which calls the "object_init_func" member (*event_init*).

The "read" functions are called by an abbreviated line:

```
(* klass->read) (&object, fp);
```

where: klass is the class included in the object structure, the object is a new GtsObject of the appropriate type (e.g., GfsMapProjection), and *fp* is a pointer to the simulation file. The GfsMapProjection class is part of the map module (modules directory), which is *distinct* from the **map.c** source file contained in the *GFS/src* directory. This module (map.mod) is loaded by function *load_module* during the keyword processing described above.

After the data have been read from the simulation file, a pointer to their memory location is stored in the (GfsSimulation *)->(GtsSListContainer *)(sim->maps) using the Glib function, *gts_container_add*.

These steps are repeated for the GfsRefine object. The members of the maxlevel structure (GfsFunction member of a GfsRefine) are not available outside of the file, **utils.c**; because of this, the (*gfs_function_**) functions are all kept in this file so that they can be accessed with a pointer to the structure and return the necessary variables. The GfsApproxProjectionParams object is read next; it consists of constant values for the variables, *tolerance* and *nitermax*.

The next GtsObject processed from the simulation file is GfsInit. This object will be examined in detail.

GfsInit Objects

The simulation file is tested for a valid input token (int, float, string, "(", and "{"). If the next token is a string, and the flag for spatially variable is set (spatial = 1), and the token is more than 3 characters long, and the input ends with **.gts**, function *read_surface* is called to read the file. If the input ends with **.cgd**, the *read_cartesian_grid* function is called.

The GfsInit is an initialization event and thus it uses functions from the GfsEventClass. The GfsEventClass was initialized by the call to *gfs_event_class* by function *gfs_classes* at the beginning of the simulation.

A GfsInit structure contains a GSList pointer (* *f*) initialized by *gfs_init* using function *gfs_init_class*(). The "read" function is initialized in *gfs_init_class_init* to be *gfs_init_read* when the GfsInit object is instantiated. The *read* function for the GfsGenericInit class has the same pointer value as the initialized "read" function in *gfs_event_class_init*, i.e. pointing to

gfs_event_read. The processing of the tide files begins with the following output from function *gfs_init_read* at run time:

```
"Begin reading...Enter gfs_init_read:...".
```

To recap, *simulation_read* calls *gfs_init_read*, which calls *gfs_event_read* as its first action.

Function *gfs_init_read* calls *gfs_event_read* to read GfsEvent objects from the simulation file. The first token read from the file is "}" by *gts_file_assign_next*. This is the end of the empty block in the simulation file; i.e., "Init {}". It then reads the "{" that signals the beginning of the next block. Control returns to *gfs_init_read*, which makes certain that the current token is "{". A loop reads the file until the closing brace, "}" is read. Carriage returns "\n" are skipped but a string must be read. Local objects are constructed to hold the data: GfsInit, GfsDomain, GfsVariable, and GfsFunction. The next 3 tokens are "A_amp", "=", and "AM2.gts".

Function *gfs_function_new* is called next to construct a GfsFunction to read the data files. This function has the same pointer value as *function_read* assigned in *gfs_function_class_init* at the beginning of the simulation. Function *gfs_function_read* is then called.

Function *gfs_function_read* is primarily a wrapper for *function_read* (see discussion above) to make the appropriate casts to use the "read" function for the requested class, which is in this case, GfsEvent (GfsInit). There is a "function_read" defined and initialized in file, utils.c. This function, GTS/*read_surface* (file utils.c) is called with the file name (e.g., **AM2.gts**) and the file pointer (GtsFile) to the simulation file.

The file, **AM2.gts**, is opened by *read_surface*. The input data are held in a GtsSurface structure that is returned to *function_read*. The (GtsSurface *s) is created in *read_surface* using functions, *gts_surface_class*, *gts_face_class*, *gts_edge_class*, and *gts_vertex_class*. These last three are all members of the GtsSurface structure. This GtsSurface is returned to *function_read* as (GfsFunction **f*)->(GtsSurface *s) from *read_surface*.

The GtsFace structure contains (GtsTriangle triangle) and (GSList *surfaces). The GtsTriangle contains (GtsEdge *e1, *e2, and *e3). These were computed from the vertices read from a ***gts*** file. Each GtsEdge contains a (GtsSegment segment) and a (GSList *triangles). Each GtsSegment contains (GtsVertex *v1 and *v2) members. Each GtsVertex contains a (GtsPoint p) and a (GSList *segments). A GtsPoint contains (gdouble x, y, and z). This can be summarized:

```
GtsFace->GtsTriangle.GtsEdge->GtsSegment.GtsVertex->GtsPoint.x, y, and z.

There are multiple GtsFace members,
  each with only one GtsTriangle,
    each with multiple GtsEdge members,
      each with only one GtsSegment,
        each of which has multiple GtsVertex members,
          for which there is only one GtsPoint,
            that is defined by x, y, and z coordinates.
```

Call function *gts_surface_read* to read the **gts** format files. Function *gts_surface_add_face* is called next to store the input surfaces in hash tables. At this point, we have confirmed that the

gts files (**bath.gts**, **AM2.gts**, and **BM2.gts**) are being read correctly by the *gts_surface_read function*, which is called by *surface_read* (**bath.gts**) or *read_surface* (**AM2.gts** and **BM2.gts**). The (GtsFace *new_face) in function *gts_surface_read* is thus inserted into the (GtsSurface *surface)->(GHashTable *faces) hash table. After all of the ordered pairs of edges describing the faces have been inserted into the hash table using their location (GtsFace *f) as the key, *gts_surface_read* returns to the calling function, *read_surface*, which does not cast the (GtsSurface *s) or even rename it. When control returns to *function_read*, however, the (GtsSurface *surface) is implicitly cast as (GfsFunction *f)->(GtsSurface *s). In other words, a valid (GfsFunction *f)->(GtsSurface *s)->(GHashTable *faces) is returned to *function_read* (file GFS/utils.c) by *read_surface*. This pointer (f->s->faces) appears to be correct.

We need to identify the class structures in which the vertex data are stored. This will be used to access the input tidal data at different locations in the code. The GTS function, *gts_surface_add_face*, is called by GTS/*gts_surface_read* to add a new (GtsFace *new_face) to the existing (GtsSurface *s)->(GHashTable *faces).

Process the surface data

When *gfs_function_read* is called by *gfs_init_read*, it receives three arguments: (1)(GfsFunction **f*); (2) (gpointer domain); and (3) (GtsFile *fp). These same arguments in *gfs_init_read* are: (GfsFunction **f*); (GfsSimulation *) from the (GtsObject **o); and (GtsFile *fp). The object in argument (2) is the (GfsSimulation *) that is returned from a call to the macro, *gfs_object_simulation*(). This implicit cast is dependent on a macro defined in GFS/simulation.h:

```
#define  gfs_object_simulation(o)  GFS_SIMULATION(GTS_OBJECT (o)->reserved)
```

Argument (2), (gpointer domain), can be returned as a (GfsSimulation *) because the parent of a GfsSimulation is (GfsDomain parent), and thus the pointer holds the correct location. The (GfsFunction *f) in *gfs_init_read* holds the face data, and the included classes are created or checked using the function, *gfs_function_new*. These are all populated with the just-read surface data as follows.

In terms of member names, we can list all of the vertex, edge, and face coordinates for a single (GtsFace *)->(GtsTriangle *triangle) in terms of x, y, and z:

```
f->triangle.e1->segment.v1->p.x
f->triangle.e1->segment.v1->p.y
f->triangle.e1->segment.v1->p.z
f->triangle.e1->segment.v2->p.x
f->triangle.e1->segment.v2->p.y
f->triangle.e1->segment.v2->p.z
f->triangle.e2->segment.v1->p.x
f->triangle.e2->segment.v1->p.y
f->triangle.e2->segment.v1->p.z
f->triangle.e2->segment.v2->p.x
f->triangle.e2->segment.v2->p.y
f->triangle.e2->segment.v2->p.z
f->triangle.e3->segment.v1->p.x
f->triangle.e3->segment.v1->p.y
f->triangle.e3->segment.v1->p.z
f->triangle.e3->segment.v2->p.x
```

```
f->triangle.e3->segment.v2->p.y
f->triangle.e3->segment.v2->p.z
```

However, only the pointer to a GfsFunction has been declared locally in *gfs_init_read*, so the members are incomplete types. This pointer (GfsFunction **f*) has the same value as in *function_read*. The data values listed above are not available in *gfs_init_read* because the full structure is not defined in this function. The data contained in a GfsFunction is copied to a GfsVariable using the function, *var_func_new* and the structure (VarFunc), both of which are in file, **event.c**.

Structure VarFunc contains (GfsVariable *v) and (GfsFunction **f*) members. Function, *var_func_new*, allocates memory for a VarFunc structure and assigns the (GfsVariable *v) and (GfsFunction **f*) input arguments to their respective members in the structure. It returns the location of this pointer (VarFunc *vf). This pointer is appended to a singly linked list called (GfsInit *init)->(GSList *f). Note that the use of the identifier, *f*, for multiple variables is a little confusing. At this point, the surface data are pointed to by the (GfsFunction *) member of a VarFunc structure but they cannot be accessed without instantiations of the necessary classes (structures GfsFunction, etc).

Function, *gfs_init_read*, reads from the simulation file (**tides.gfs**) until a closing "}" is reached. Each GfsInit object (i.e., "**AM2.gts**", "**BM2.gts**", and "flip") is read and placed in a local VarFunc structure, before being appended to (GfsInit *init)->(GsList **f*).

Insert data values into GtsContainers

The GtsSurface data are contained in a GfsInit structure. This GfsInit structure contains the pointers to the hash tables containing the surface data in a singly linked list (GSList *f). The parent of a GfsInit object is a GfsGenericInit object, which is a synonym (i.e., *typedef*) for a GfsEvent. This GfsInit structure must be locatable within the GfsSimulation structure. The GfsSimulation class includes a GtsSListContainer pointer called "events". The "Init" events are contained within the GtsContainer named "events". They already exist and only need to be pointed to.

The connection (between the already-extant face data and the simulation) is completed by function *gts_container_add*, which is called to add the surface data to the simulation structure. The first argument to *gts_container_add* is (GfsSimulation *sim)->(GtsSListContainer *events), which is implicitly cast to (GtsContainer c) when passed to *gts_container_add*. The second argument passed to *gts_container_add* is a (GfsInit *object), which is implicitly cast to a (GtsContainee *) by use of the macro, GTS_CONTAINEE. The (GfsInit *object) *cum* (GtsContainee *) will be placed in the (GtsSListContainer *events) member of the (GfsSimulation *sim) structure. The GtsContainee class has a GtsObject parent.

The (GfsInit *)->(GtsSListContainee *) passed to *gts_container_add* as (GtsContainee) is added to the (GfsSimulation *sim) *cum* (GtsContainer) using functions, *slist_container_add*, *container_add*, and *slist_containee_add_container*. Function, *gts_container_add*, is an entry point to permanently store the surface data. A new GtsContainer object is constructed if necessary.

During object instantiation, the (GtsContainerClass *) parent of the (GtsSListContainer *c) is also initialized. Its initialization function is *container_class_init*. This function sets

(GtsContainer *klass)->(void *add) member to be *container_add*. When *slist_container_add* calls the "add" member of its parent, it is referring to function, *container_add*. This has been verified with print statements. The value of the passed (GtsContainee *item) is the same as well. Function, *container_add*, calls the "add_container" function for a GtsSListContainee object, which is its parent. Following the general trend of an "SListContainee" class having a "Containee" class for a parent, we find that the GtsContaineeClass has an "add_container" member. However, this is where the standard method is a little different.

The "add_container" function initialized in *containee_class_init* is NULL. Instead, the *slist_containee_class_init* function assigns "add_container" to be *slist_containee_add_container* after casting the (GtsSListContaineeClass *klass) to be a GtsContaineeClass using the macro, GTS_CONTAINEE_CLASS. This function is called by *container_add*. Note that *container_add* reverses the order of the arguments it receives before calling *slist_containee_add_container*. Function, *slist_containee_add_container*, prepends the (GtsContainee *i) pointer to the (GtsContainer *c) list. This is the same value as the (GfsInit *object) originally passed from function, *simulation_read*.

Activate GfsEvents

The initialization function for GfsEvents is *gfs_event_init*. This function is not directly called from *ocean_run*. There are wrapper "foreach" functions that will loop over all of the GfsEvents associated with the GfsSimulation.

Function, *ocean_run* passes three arguments to *gts_container_foreach*, which are map to dummy arguments as follows:

1. GTS_CONTAINER ((GfsSimulation *sim)->(GtsSListContainer *events)) => (GtsContainer *c)
2. ((GtsFunc) *gfs_event_init*) => (GtsFunc *func*)
3. (GfsSimulation *sim) => (gpointer data).

The (GtsContainer *c) is a pointer to the (GtsSListContainer *events) containing the GSList of pointers to the faces read from the gts files (e.g. **AM2.gts** and **BM2.gts**). (GtsFunc *func*) is a pointer to the user-supplied function, which for event initialization is *gfs_event_init*. This has been verified with pointers to this function. The (gpointer data) is a pointer to the GfsSimulation structure. Function, *gts_container_foreach*, is a wrapper for a user-supplied "foreach" function specific to the class. This "foreach" is identified using the following code:

```
(* GTS_CONTAINER_CLASS (GTS_OBJECT (c)->klass)->foreach) (c, func, data);
```

We know the class is a GfsInit because it has been printed in function, *gfs_event_init*, with the same pointer value as (GSList *events) from *simulation_read*. The default class is GfsEventClass, which has no "foreach" member. Note, however, that GfsEventClass does contain an "event" member, which is initialized as "klass->event = *gfs_event_event*" in function, *gfs_event_class_init*. The parent is (GtsSListContaineeClass *parent_class), which has a (GtsContaineeClass) parent. The GtsContaineeClass structure has a "foreach" member, which is initialized to NULL in function, *containee_class_init*.

The GfsInit class (GfsEventClass) is cast to a GtsContainerClass using the macro, GTS_CONTAINER_CLASS. The GtsContainerClass structure also has a "foreach" member,

which is initialized to NULL in function, *container_class_init*. The GtsSListContainerClass and GtsSListContaineeClass classes both have "foreach" equal to *slist_containee_foreach*. The (GtsContainer *c) received by *slist_container_foreach* (Argument 1) is (GfsSimulation *sim)->(GtsSListContainer *events), which was cast to a GtsContainer in *ocean_run* before it was passed to *gts_container_foreach*.

The function to be substituted to the "foreach" function is passed to *gts_container_foreach*. For example, the GfsInit class under consideration has the function name, "gfs_event_init" hardwired in the call to *gts_container_foreach* in *ocean_run*. Thus, when *slist_container_foreach* is called by gts_container_foreach, it applies this function in the "while" loop over all items in the passed container,

```
ocean_run
  call gts_container_foreach
    call slist_container_foreach
      loop over GfsEvents
        call gfs_event_init
        call gfs_event_event
```

GfsInit events are activated by function, *gfs_event_event*, in the following line:

```
(* GFS_EVENT_CLASS (GTS_OBJECT (event)->klass)->event) (event, sim);
```

This last function, *gfs_event_event*, activates the initial events as well as the recurring ones. In this case, "activated" refers to setting (GfsEvent *event)->(gboolean realized) true. This was verified with print statements.

Loop over all (GfsDomain * domain)->(GSList * variables) to initialize. A while loop examines all of the variables in the (GSList *variables), and activates them using *gfs_event_init*. Function, *gfs_domain_bc*, is called to initialize the boundary conditions in @domain using function, *gfs_domain_copy_bc*. The boundary conditions are of specific interest and will be examined further. For now we note the following hierarchy in function, *ocean_run*:

```
(GfsSimulation *sim)->(GfsDomain *domain)->(GSList * variables)

struct GSList {
  gpointer data;
  GSList *next;
};
```

We loop over the items in the variables list by:

```
GSList * i = domain->variables;
while (i) {
  gfs_event_init (i->data, sim);
  gfs_domain_bc (domain, FTT_TRAVERSE_LEAFS, -1, i->data);
  if (GFS_IS_VARIABLE_RESIDUAL (i->data))
    res = i->data;
  i = i->next;
}
```

Open Boundary Conditions on a GfsBox

The first operation in *bc_value_read* is to call the *read* function for its parent. The parent of a GfsBcValue object is a GfsBc structure; the *read* function for this class is *bc_read*, which parses the "U" and "0" strings from the simulation file but does not construct a function. It contains no calls to *gfs_function_read*. It does create a (GfsBc->variable) member (v) from "U" and the domain variables. Nothing is done with the "0" character read from the file. Control returns to *bc_value_read*, which immediately calls *gfs_function_read* to generate a GfsFunction object from whatever is parsed next from the simulation file.

The "0" that was read from the simulation file is cast to a float in *function_read* and the "H" is read before control returns to *gfs_function_read*, and then to *bc_value_read*. Control then returns to *bc_flather_read* (file ocean.c), which reads the "P" from the simulation file. This token must be "P" or an error is incurred. A variable is added to the GfsBc structure from this name and the variable list for the grid (GfsDomain *domain)->(GsList * variables).

The tidal amplitude function is read with the following replacement for "M2(t)":

```
(A_amp*cos (M2F*t)+B_amp*sin (M2F*t))
```

Thus, the next token read from the file in *bc_flather_read* is "(". A new GfsFunction object is created by a call to *gfs_function_new* and passed to *gfs_function_read*, which then calls *function_read*. This line is parsed into a new (GfsFunction *f), which is a synonym for non-static member functions implemented as Gerris Plug-ins (Section 3). The first step is a call to *gfs_function_expression*, which produces the following string and returns it to *function_read* as a GString:

(A_amp*cos ((2.*M_PI/44700.)*(t))+B_amp*sin ((2.*M_PI/44700.)*(t)))

This is assigned to the *expr* member of the GfsFunction (i.e., f->expr) that was created when *function_read* was entered. This is discussed further in the time-dependent GfsFunction example.

References

Popinet S. 2003. Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries. Journal of Computational Physics 190:572-600.

Popinet S, Zaleski S. 1999. A front-tracking algorithm for accurate representation of surface tension. International Journal for Numerical Methods in Fluids 30:775-93.

Appendix B. GNU Triangulated Surface (GTS) Library

Introduction

The Gnu Triangulated Surface (GTS) Library is open source free software intended to provide a set of useful functions to deal with 3D surfaces meshed with interconnected triangles. The fundamental process used in the GTS library is Delaunay triangulation (Figure B.1).

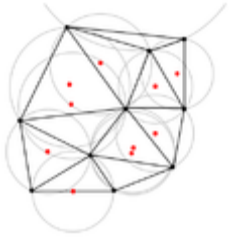


Figure B.1. A Delaunay triangulation of a surface.

The GTS library is built upon the base class, GtsObjectClass (Figure B.2). This system includes the geometric entities used to construct surfaces as well as utilities such as file processing.

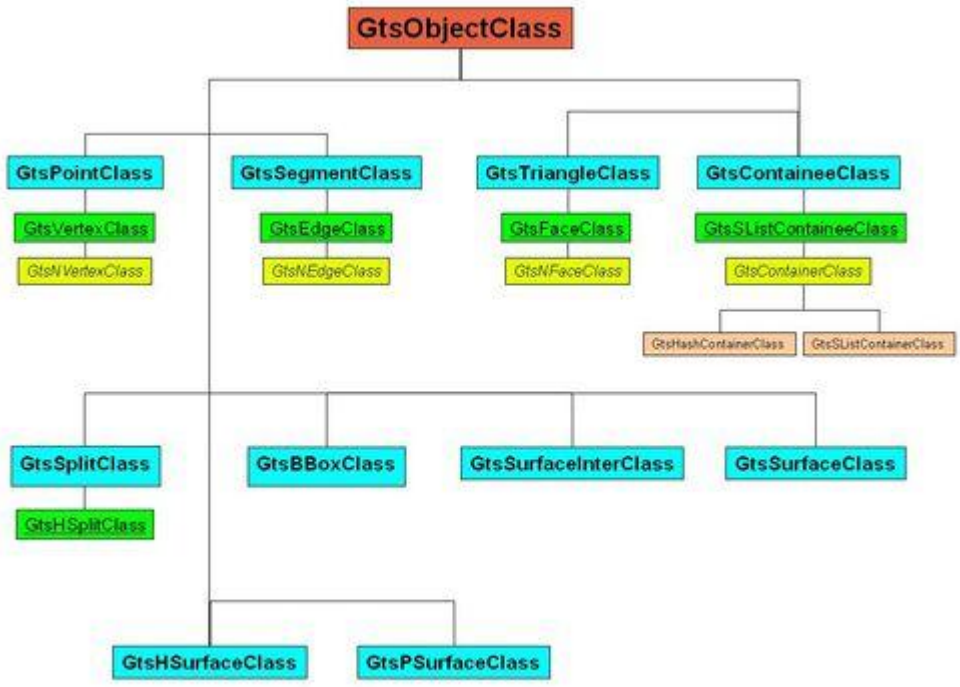


Figure B.2. Diagram of the basic classes within GTS.

Some of these classes should be recognizable; for example, points and segments are used to construct triangles, which are then used to build surfaces. The subclasses of these classes (shown in green) represent these geometrical relationships. The GtsSurfaceClass will be discussed in more detail below. These classes are implemented using *c structures*.

GTS Objects, Classes, Constructors, and Inheritance

The base class for all GTS and GFS objects is the GtsObjectClass (Figure B.3). This class defines generic functions used in the libraries: *clone*; *destroy*; *read*; *write*; *color*; and *attributes*. These functions have standard arguments; the read function requries a (GtsObject **), which is a pointer to an array of objects, and a GtsFile pointer. The actual name of the function is assigned in the *GtsObjectClassInitFunc* function supplied by the user for each class. For the GfsOcean object this is *gfs_ocean_class_init*. This function assigns *ocean_read* to the variable, read.

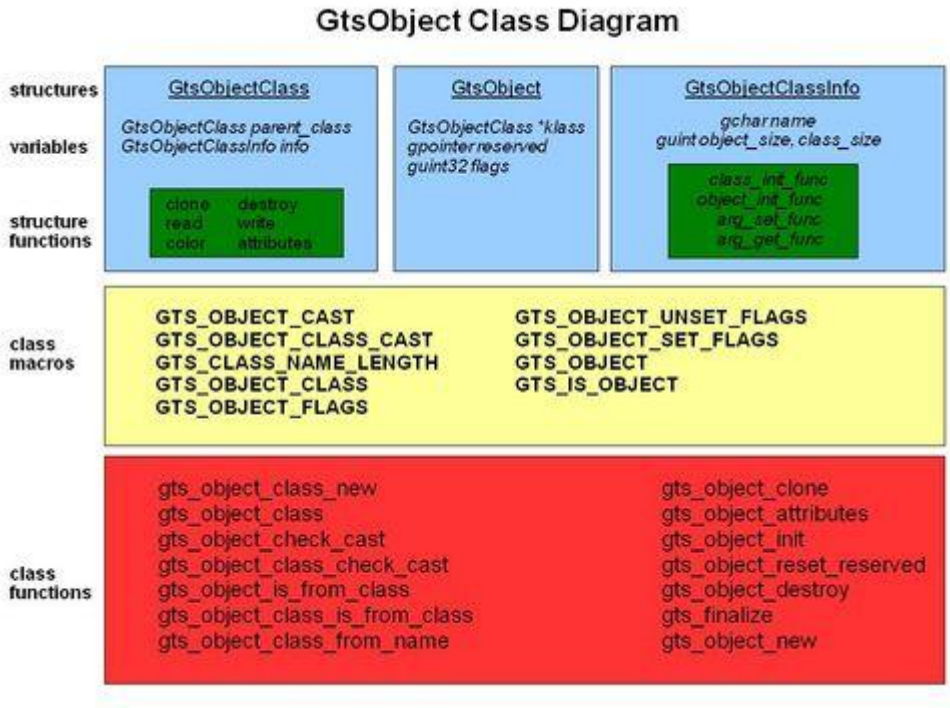


Figure B.3. Class diagram for the GtsObject. The blue boxes contain variables and functions for the class structures (green). C preprocessor macros are listed in yellow boxes, and class functions are contained in red boxes.

The structure, GtsObjectClassInfo contains (among others) a GtsObjectClassInitFunc named *class_init_func*. This structure is the first member of the GtsObjectClass and thus these special functions are included in every object class like GfsOcean. The functions that are used by a class are initialized in the *class_init_func* member of the GtsObjectClassInfo structure. This "info" structure was initialized in *gfs_ocean_class* (*gfs_init* at start of *main*) for this object.

Parent classes are automatically constructed when any class is created. This is completed by *gts_object_class_init*. This is done because this is the base class for all classes. It guarantees that the necessary members have been created for all new class structures. Thus, when *gts_graph_read* recognizes a GfsOcean class, it can assign a value to *klass* and make certain that the necessary included functions are ready. The GtsObjectClass named "GfsOcean" was created by *gfs_init*. There is redundancy here to fit the object-oriented concept but the functions all have checks that assure it will be error free.

The procedure for implementing a new class is as follows:

1. Call a class function like *gts_slist_container_class*(), which returns a pointer to a structure for that class (e.g., GtsSListContainerClass).
2. This function has a static pointer variable to the desired class structure that will be available within this function only. If it has been assigned, nothing happens here.
3. Initiate the GtsObjectClassInfo member of the GtsObjectClass that is part of the GTS library with a name for the class, its size, and init functions as declared in the GTS header file, gts.h.
4. Call *gts_object_class_new* with the parent class constructor function (e.g., *gts_container_class*) and the "info" structure as args. It returns a pointer to the new class structure.
5. The function, *gts_object_class_new*, uses the "info" structure to assign the memory locations of the "init" functions. The parent constructor will return a pointer to the parent class, which will be passed to *gts_object_class_new*.
6. Part of the function of *gts_object_class_new* is to instantiate the requested class using the init function (e.g., *slist_container_class_init*).
7. The "init" function defines function names that are consistent with the parent class. The example defines: add = slist_container_add, remove=slist_container_remove, foreach=slist_container_foreach, and size=slist_container_size (members of the GtsContainerClass structure defined in gts.h), and an additional function, destroy=slist_container_destroy.

GtsObjectClass

The most basic macros, classes, and functions are associated with the GTS library. Many are defined in the header file, gts.h. This can be demonstrated using the basic class, GtsObject, which is defined as a structure in gts.h:

```
typedef struct _GtsObject        GtsObject;/* line 69  in gts.h   */
```

The basic structure (class), _GtsObject, is not used in other declarations; instead, GtsObject is used. The object structure contains a pointer to a class (klass) of which it is a member. This will be used often for user-defined classes and objects, which must be consistent.

A structure of class GtsObject includes information about the class itself. It thus includes a pointer to a structure of class GtsObjectClass, which includes listings of dummy functions for copying, etc. its members, as well as a GtsObjectClassInfo structure as a member (info), and a pointer to the GtsObjectClass of the class' parent (parent). The info member is a structure

containing the class name, object and class sizes, and dummy functions to initialize the object and its parent class, as well as setting and getting other arguments related to the object. Here are the listings of these structures, which represent classes in an object-oriented framework:

```
struct _GtsObject {                                    /* line 235 in gts.h
      */
  GtsObjectClass * klass;
  gpointer reserved;
  guint32 flags;
};
struct _GtsObjectClass {                               /* line 242 in gts.h
      */
  GtsObjectClassInfo info;
  GtsObjectClass * parent_class;
  void        (* clone)      (GtsObject *, GtsObject *);
  void        (* destroy)    (GtsObject *);
  void        (* read)       (GtsObject **, GtsFile *);
  void        (* write)      (GtsObject *, FILE *);
  GtsColor    (* color)      (GtsObject *);
  void        (* attributes) (GtsObject *, GtsObject *);
};
struct _GtsObjectClassInfo {
  gchar name[GTS_CLASS_NAME_LENGTH];
  guint object_size;
  guint class_size;
  GtsObjectClassInitFunc class_init_func;
  GtsObjectInitFunc object_init_func;
  GtsArgSetFunc arg_set_func;
  GtsArgGetFunc arg_get_func;
};
```

Initializing a New GtsObject

This procedure is repeated here because it is complex and almost always used. This section is referenced from several locations in this document with the name of the new class changed.

(1) The (GfsInit *object) passed from simulation_read is implicitly cast as a (GtsContainer *c) because it is a dummy argument in *gts_container_add*.

(2) The macro, GTS_OBJECT, is invoked for the actual pointer value. This macro is a call to *gts_object_cast*, with a pointer to the GtsInit parent class as returned from the function, *gts_object_class*.

```
#define GTS_OBJECT(obj)  \
        GTS_OBJECT_CAST (obj, GtsObject, gts_object_class ())
#define GTS_OBJECT_CAST (obj, type, klass)  \
        ((type *) gts_object_check_cast (obj, klass))
```

(3) The macro, GTS_CONTAINER_CLASS, is passed the parent class, which in long form is (GtsContainer *c)->(GtsSListContainee object).(GtsContainee containee).(GtsContainee object). (GtsObject object).(GtsObjectClass *klass); this is abbreviated in the source code as, GTS_OBJECT(c)->klass, because the intermediate parents are all contained rather than using

pointers. A print statement verifies that: (GTS_CONTAINER_CLASS (GTS_OBJECT (c)->klass)->parent_class.parent_class.parent_class.info.name)) equals "GtsSListContainer".

(4) Function *gts_object_class* checks that the base class GtsObjectClass is present. This class pointer is then passed to the analogous macro, GTS_CONTAINER_CLASS. It then passes the returned (GtsContainerClass *) pointer from function, *gts_container_class*, to macro, GTS_OBJECT_CLASS_CAST.

```
#define GTS_OBJECT_CLASS_CAST (objklass, type, klass) \
    ((type *) gts_object_class_check_cast (objklass, klass))
```

(5) Function, *gts_container_class*, initializes the "info" structure member "GtsObjectClassInitFunc" to be *container_class_init*. It then calls *gts_object_class_new* with the (GtsSListContaineeClass *) returned from function, *gts_slist_containee_class*; this is the first argument to GTS_OBJECT_CLASS.

```
#define GTS_OBJECT_CLASS (klass)  \
   GTS_OBJECT_CLASS_CAST (klass, GtsObjectClass, gts_object_class())
```

This iterative method assures that all required parent classes exist.

(6) Function, *gts_slist_containee_class*, initializes the (GtsSListContaineeClass *)->info member, GtsObjectClassInitFunc, to be *slist_containee_class_init*. This is done only if this class does not exist already. Function, *gts_object_class_new*, is called to allocate memory and call *gts_object_class_init*, which checks for all parents all the way to the GtsObjectClass and calls their *class_init_func* members if needed.

(7) The *add* function is a member of the GtsContainerClass structure, which is the parent of a GtsSListContainerclass. It is initialized in function, *slist_container_class_init*, to be *slist_container_add*. Thus, when the GtsSListContainer is cast as a GtsContainerClass, this add function is called.

The trivial case for no type checking could lead to errors at run time. If GTS_CHECK_CASTS is defined, the rules for macro substitution indicate that "obj" will be passed directly to GTS_OBJECT_CAST. However, GtsObject and gts_object_class() are not defined in the macro. These strings will be substituted into any calls of GTS_OBJECT; for example, GTS_OBJECT(s) becomes GTS_OBJECT_CAST(s, GtsObject, gts_object_class()). GtsObject is defined in file gts.h, and thus is known to be a (struct GtsObject), which is really a class. This macro results in the following substitution from the original GTS_OBJECT:

```
((GtsObject *) gts_object_check_cast (obj, gts_object_class ())
```

The function gts_object_check_cast() checks that obj exists and can be cast to the return value from the function gts_object_class(), but is not from this class. In other words, the function gts_object_class() is executed before the types are checked. The function, gts_object_class(), which initializes the structure for a gts_object_class, is defined in gts-0.7.6/src/object.c as:

```
/**
* gts_object_class:
*
* Returns: the #GtsObjectClass.
*/
```

```
GtsObjectClass * gts_object_class (void)
{
 static GtsObjectClass * klass = NULL;
  if (klass == NULL) {
    GtsObjectClassInfo object_info = {
      "GtsObject",
      sizeof (GtsObject),
     sizeof (GtsObjectClass),
      (GtsObjectClassInitFunc) object_class_init,
      (GtsObjectInitFunc) object_init,
      (GtsArgSetFunc) NULL,
      (GtsArgGetFunc) NULL
    };
    klass = gts_object_class_new (NULL, &object_info);
  }
  return klass;
}
```

There are no args passed to this function because it automatically generates an object of class, “GtsObject”. The sequence of initializing a new base object using GTS is as follows:

1. call gts_object_class (): Initialize a new GtsObjectClassInfo.object_info structure including replacing the dummy function class_init_func with object_class_init.
2. call gts_object_class_new (); Check for info and parent; Initialize a hash table with g_hash_table_new and place the pointer to the class structure in it using the object class name as the key. The hash table pointer (class_table) is only accessible to the functions in file object.c because it is static within this file.
3. Call gts_object_class_init (); This function is expecting (GtsObjectClass *) args for both the new class and the parent class, but it is passed the current class (klass) for both args. It is recursive as long as the parent class has a parent class itself. Since klass->parent_class is NULL, the recursive call results in an immediate return and the next statement is executed.
4. Call class_init_func (); This is a dummy function that does nothing; it is only defined for generality. When the structure GtsObjectClassInfo is initialized, this dummy is replaced with object_class_init, which initializes the functions within the class for clone, destroy, read, write, color, and attributes.

When this sequence has completed, an object of class 'GtsObject' has been generated along with its basic members. The object-oriented concept of *inheritance* is represented by the inclusion of parent classes (pointers to their structures) within a class structure. This is represented in the diagram above by the yellow boxes for the lower classes, which actually include classes from the green boxes, which in-turn include classes from the blue boxes. Ultimately, all classes (structures) inherit the GtsObjectClass and all of its members.

GtsSurfaceClass

As described in the Introduction, the triangulated surface is a basic construction for the finite-volume method implemented in Gerris to conserve mass. These surfaces are used for the

coastline, the seafloor, and boundary condition like a water surface anomaly computed from tidal constituents. We will be discussing these applications in other sections.

Class Structure

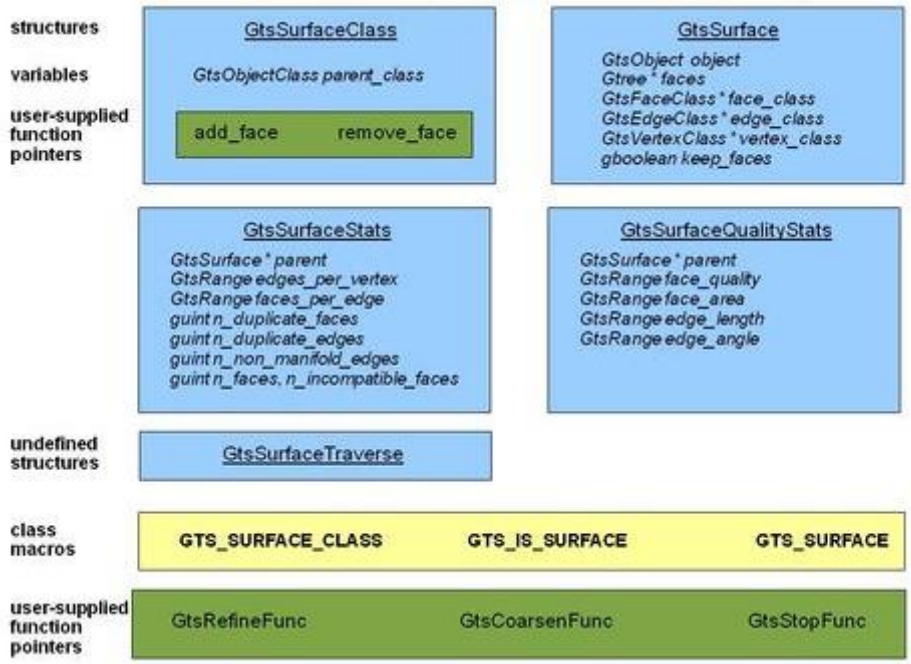
The basic functionality of a surface is introduced through the GtsSurfaceClass:

```
 typedef struct _GtsSurface       GtsSurface;         /* line 85 of gts.h  */

 struct _GtsSurface {                                 /*  line  1062  of  gts.h
*/
   GtsObject object;
 #ifdef USE_SURFACE_BTREE
   GTree * faces;
 #else /* not USE_SURFACE_BTREE */
   GHashTable * faces;
 #endif /* not USE_SURFACE_BTREE */
   GtsFaceClass * face_class;
   GtsEdgeClass * edge_class;
   GtsVertexClass * vertex_class;
   gboolean keep_faces;
 };
```

The *C* structure, GtsObject, is a member of class GtsSurface. This means that a GtsSurface includes the information contained in the GtsObject class as well as data about the surface; i.e., its face, edge, and vertex description contained in objects of the GtsFaceClass, GtsEdgeClass, and GtsVertexClass classes (Figure B.4A). Note that a GtsFace, GtsEdge, and GtsVertex are all generated in an analogous manner to the GtsObject class described above. It also means that the a GtsSurface class includes the same data as a GtsObject class (inheritance).

Figure B.4. Class diagrams for surfaces.



A. Simplified class diagram for the GtsSurface Class.

B. Member functions for the GtsSurface class.

The large number of functions associated with this class indicates the level to which surface operations have been modularized (Figure 4B). These functions are not modified by a user, however, but they can be implemented for user-defined operations.

The GtsSurface structure contains a GHashTable * faces member. The hash table algorithm does not use the surface binary tree. This means that a hash table is used instead of a tree unless the macro USE_SURFACE_BTREE is set). Both are available in glib. The hash table is

initialized with *g_hash_table_new* and the tree with *g_tree_new*. There is a useful tutorial for the hash table library in glib. The surface->faces member is a new hash table with no entries (NULL, NULL).

Creating a GtsSurface

Generating a new GtsSurface is a two-step process; first, a class must be created and then a surface object is made every time an operation changes an existing one or instantiates one.

- call gts_surface_class (); Initialize new GtsObjectClassInfo.surface_info structure, including replacing the dummy function class_init_func with surface_class_init, and object_init_func with surface_init.
  a. Call gts_object_class_new (gts_object_class (), &surface_info). Note that the embedded function gts_object_class will generate a new GtsObjectClass. The function, surface_class_init (listed in GtsObjectClassInfo) declares surface_destroy, surface_write, add_face, and remove_face function names.
  b. Return a pointer to the new GtsSurfaceClass.

The static GtsSurfaceClass pointer, klass, in function *gts_surface_class* is used to prevent the generation of multiple surface classes because this would destroy information previously acquired. After a GtsSurface class has been instantiated, new surface objects are generated as computations are carried out because surfaces are constantly changing in response to the grid adapting. Thus, the *gts_surface_new* function is not called by *gts_surface_class* as was done for a new object class.

The following list shows functions contained within file .../gts.../src/surface.c that call gts_surface_new:

- static void *traverse_boundary* (GtsEdge * e, gpointer * data)
- static void *traverse_remaining* (GtsFace * f, gpointer * data)

There is a file called *surface.c* that is part of gerris. This is a bad idea but it is included in a different location so they did it. It does not repeat .../gts.../src/surface.c but supplements it with new classes. It also calls the function, gts_surface_new from the following functions:

- static void *surface_read* (GtsObject ** o, GtsFile * fp)
- static void *face_overlaps_box* (GtsTriangle * t, gpointer * data)

To continue with generating a new GtsSurface:

```
call gts_surface_new (        GtsSurfaceClass        * klass,
                              GtsFaceClass   * face_class,
                              GtsEdgeClass   * edge_class,
                              GtsVertexClass * vertex_class)
```

For all examples of this function, gts_surface_class is called (see above) from the argument list either directly (gfs_surface.c) or using macros defined in gts.h., but it does nothing if klass is not NULL. The instantiation of a (GtsSurface *) at line 137 of file .../gts../surface.c first calls GTS_OBJECT_CLASS(klass). The instantiations in *surface.c* checks that the requested object can be cast to the current value of klass. The cases in *gfs_surface.c* generate GtsSurface objects directly and do not need to complete this check. The macro GTS_SURFACE is used to check for compatibility of the new surface's parent object using the macro GTS_OBJECT_CLASS with the current klass object in the argument to the function gts_object_new. This name is similar but this function has not been invoked in generating a new GtsObject above.

Call function gts_object_new

- Allocate memory using data in structure info.
  - Call gts_object_init, which calls the object_init_func function (initialized as object_init in structure GtsObjectClassInfo), which initializes only the reserved and flags parameters. This is done recursively as with the GtsObjects.
  - Return a new surface object of klass, 'GtsSurface'.

Before a new surface object can be generated, however, the required face, edge, and vertex classes need to be instantiated. This is because surfaces are only generated by some operations as listed above and they are defined by their boundaries. I have tested this; the test printed the following output for the new object and its classes:

```
object_1_ptr->object.klass->info.name = GtsSurface
object_1_ptr->object.klass->parent_class->info.name = GtsObject
object_1_ptr->face_class->parent_class.parent_class.info.name = GtsFace
object_1_ptr->edge_class->parent_class.parent_class.info.name = GtsEdge
object_1_ptr->vertex_class->parent_class.parent_class.info.name = GtsVertex
```

Reading a GtsSurface from a File

The reading functionality has been tested for bathymetry and tidal elevation data for the Mississippi Bight. This analysis was motivated by problems I had when trying to read tidal data from files. There are three files read for this simulation: **bath.gts**; **AM2.gts**; and **BM2.gts**. File, **bath.gts**, is opened in *surface_read* (GFS/surface.c). This operation occurs twice with different file pointers generated. The first time occurs after *gfs_solid_read* and the second time appears to be after *gfs_solid_class*. **AM2.gts** and **BM2.gts** are opened in *read_surface* (GFS/utils.c). No wonder this is confusing--the bath file (**bath.gts**) is opened in *surface_read* (GFS/surface.c).

The GTS file format is as follows:

```
* All the lines beginning with #GTS_COMMENTS are ignored. The first line
* contains three unsigned integers separated by spaces. The first
* integer is the number of vertices, nv, the second is the number of
* edges, ne and the third is the number of faces, nf.
*
* Follows nv lines containing the x, y and z coordinates of the
* vertices.  Follows ne lines containing the two indices (starting
```

```
* from one) of the vertices of each edge. Follows nf lines containing
* the three ordered indices (also starting from one) of the edges of
* each face.
*
* The format described above is the least common denominator to all
* GTS files.  Consistent with an object-oriented approach, the GTS
* file format is extensible. Each of the lines of the file can be
* extended with user-specific attributes accessible through the
* read() and write() virtual methods of each of the objects written
* (surface, vertices, edges or faces). When read with different
* object classes, these extra attributes are just ignored.
```

The *read* functions call the same GTS library functions, however, and thus read a **gts** formatted file. Here is an example of a simple surface, which requires a very small file to describe:

```
4 5 2 GtsSurface GtsFace GtsEdge GtsVertex
270.5 29.5 -20
270 29.5 0
270 29 0
270.5 29 -20
1 2
1 3
3 2
4 1
4 3
1 2 3
2 4 5
```

The first line lists the number of vertices, edges, and faces that describe the surface. The vertices are read first.

A vertex class is created with the macro, GTS_VERTEX, following the usual operation. A GtsVertex (Figure B.5) consists of the *x*, *y*, and *z* coordinates of a point. The GtsVertex structure contains a GtsPoint structure (*p*) and a GSList (*segments*). A temporary GtsObject, *new_vertex*, holds the *x, y*, and *z* values returned by the vertex_class read function, *point_read*, which is the *read* function for GtsPointClass (initialized in *point_class_init*). This is a generic function to read a point triplet from a file. Function, *point_read*, places the triplet it has read into the (gdouble) x, y, and z members of the GtsPoint structure. The vertices are then assigned to the *n*th element of a GtsVertex array (vertices[*nv*]). Note that the GSList pointing to these in order is called *segments*.

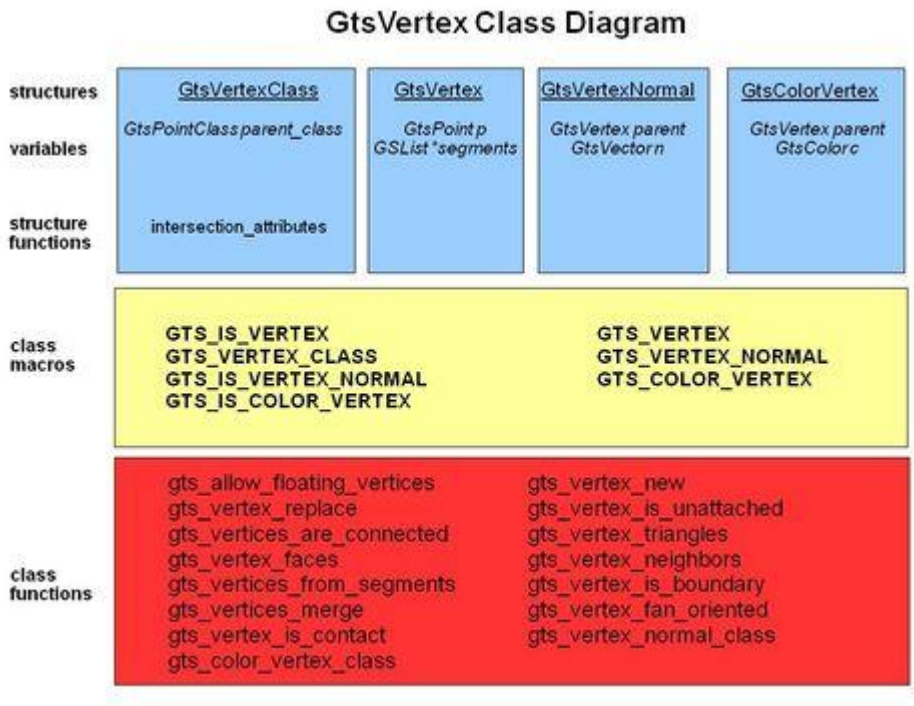


Figure B.5. Diagram of the pseudo-class, GtsVertex.

The edges are listed next as pairs of vertices in the **gts** file. A (GtsEdge * new_edge) is assigned based on the values for the *vertices*[*nv*] read from a **gts** file using *gts_edge_new*. The GtsEdge array (*edges*[*ne*]) contains pointers to each edge read from the file. These ordered indices are read by *gts_surface_read* and placed in GtsEdge structures (Figure B.6) as GtsSegments (*segment*). The GtsSegment class contains GtsVertex pointers to the two vertices that describe an edge. Note that the GSList pointing to these is called *triangles*.

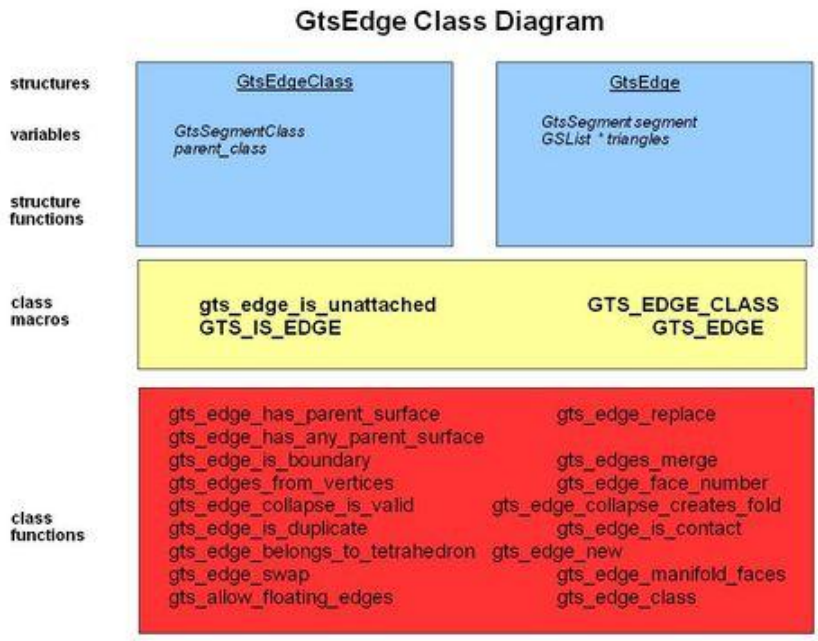


Figure B.6. Diagram of the pseudo-class, GtsEdge.

The faces of the surface are read as point triplets to create a GtsFace object (Figure B.7) using function, *gts_face_new*. This loop is also used to add the new face to the (GtsSurface *surface)->(GHashTable *faces) hash table using *gts_surface_add_face*, which is a wrapper for the Glib function, *g_hash_table_insert*.

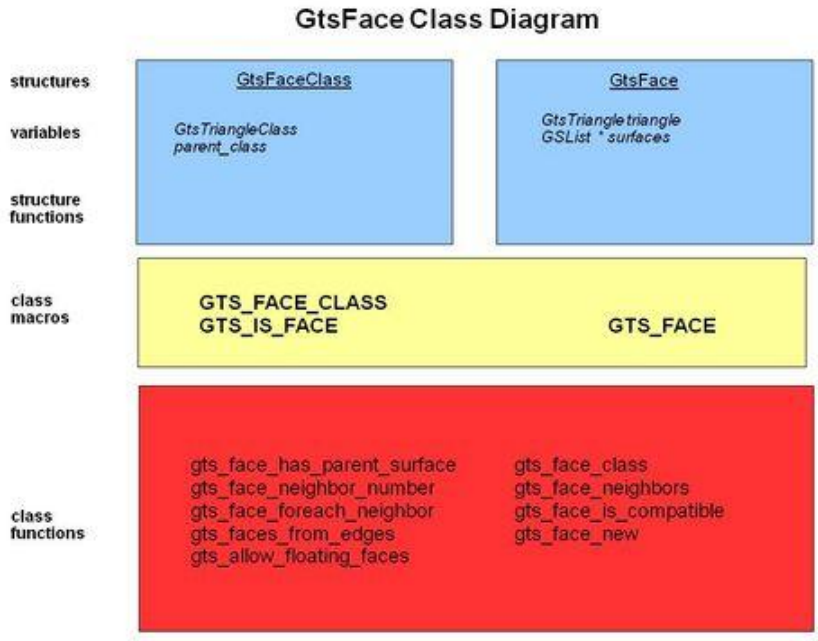


Figure B.7. Diagram of the pseudo-class, GtsFace.

These triples are assigned to a new GtsFace object, * new_face, using function *gts_face_new*, which is passed the *edges*[*ne*] array. Function *gts_face_new* initializes a new GtsFace (if necessary) and calls *gts_triangle_set* to construct a triangle from a triple of edges. After the current face is read from the file and its structures populated, *gts_surface_add_face* is called with the (GtsSurface * surface) and the (GtsFace * new_face) arguments. Function *gts_surface_add_face* checks that the hashtable key (GtsFace *f) passed as an argument is associated with a value in the (GHashTable * faces) member of the GtsSurface *. If not (i.e., NULL is set during initialization of the table), the current (GtsSurface *s) is prepended to the (GSList *surfaces) member of (GtsFace *f). The (GHashTable *) member of (GtsSurface s)->faces is updated with the current (GtsFace *f) inserted as the key and the value; i.e., *g_hash_table_insert* (s->faces, f, f). After updating the hash table, the *add_face member* of the GtsSurfaceClass structure is called with arguments, (GtsSurface *s, GtsFace *f).

GtsGraph Class

The GtsGraph structure (aka class) is part of the GTS library. It bridges the geometric gap between a mathematical graph, a GtsContainer, a GtsHashTable, and input vector data (e.g., bathymetry). The GtsGraph is included in the GTS library as a container for general domain information (e.g., number of nodes and boxes), as well as containing other structures for the graph data read from the ***.gts** files by function, *gts_graph_read*. This class is represented by the following *C* structures:

```
struct _GtsGraphClass {
  GtsHashContainerClass parent_class;
  gfloat (* weight) (GtsGraph *);
};

struct _GtsGraph {
  GtsHashContainer object;
  GtsGNodeClass * node_class;
  GtsGEdgeClass * edge_class;
};
```

This is a utility that is not actually used by the GTS library itself. All of the usual GtsObject properties are included through the following inheritance path:

```
(GtsGraphClass                                                    *)-
>GtsHashContainerClass.GtsContainerClass.GtsSListContaineeClass.  \
                                    GtsContaineeClass.GtsObjectClass.read
```

In this pseudocode, each consecutive parent class is named rather than listed as a member of the child class. The functions are all assigned to the base class, the GtsObjectClass. This is an important lineage for understanding the way Gerris (and GTS) processes gridded fields. This is a generic structure that can be filled with any type of data through its GtsHashContainer member. It is used at the top level for the Gerris domain because it includes the most basic topological information--the number of nodes and edges in the simulation. An important distinction between this "graph" and the vector data (i.e. "graph") read from the ***.gts** files is

that there are no location data associated with the GtsGraph. That is, it contains information about the nodes in the simulation but no dimensional data at all.

A new GtsGraph object is created by the following line:

```
(1)  g = GTS_GRAPH (gts_object_new (GTS_OBJECT_CLASS (gts_graph_class ())));
```

The result of this line is instantiations of the following (classes) structures by the *gts_graph_read* function (including parents):

```
 GtsGraph:
   GtsHashcontainer         object
     GtsContainer           c
     GHashTable *           items
     boolean                frozen
   GtsGraphClass *          graph_class
     GtsHashContainerClass  parent_class
     gfloat                 (* weight) (GtsGraph *)
   GtsGNodeClass *          node_class
     GtsSListContainerClass parent_class
     gfloat                 (* weight) (GtsGraph *)
   GtsGEdgeClass *          edge_class
     GtsContaineeClass      parent_class
     GtsGEdge *                (* link) (GtsGEdge *e, GtsGNode *n1, GtsGNode
*n2)
     gfloat                 (* weight) (GtsGEdge *e)
     void                   (* write ) (GtsGEdge *e, FILE * fp)
```

The (GHashTable *items) is a pointer to any data associated with the GtsGraph. The function *gts_hash_container_class* in-turn initializes a GtsContainerClass and assigns the *hash_container_add*, **_remove*, **_foreach*, and **_size* functions. These are part of the GTS library. They use the glib c commands for hash containers (e.g., *g_hash_table_insert*). It is important to note that the GtsGraph pointer, *g*, that has been created by line (1) is returned by the function, *gts_graph_read*. It is going to be the index to locate the vertex data elsewhere.

The GtsWGraph class encapsulates a GtsGraph and adds a weight variable:

```
 GtsWGraph:
   GtsGraph  graph
   gfloat    weight
```

The function *gts_graph_class*() creates a GtsGraphClass and initializes the graph_info structure (including *init* and *read* function names). All of the GTS objects are instantiated as GtsObjects; for example, the function, *gts_hash_container_class*, is invoked from within the macro, GTS_OBJECT_CLASS, which will initialize a new class and create (if necessary) a new parent class for the hash container class, a GtsContainerClass. This recursive sequence will produce class pointers for all of these classes.

Function, *gts_object_class_new*, is called in *gts_graph_class* with the GtsHashContainerClass pointer cast as a GtsObject and passed as its first argument. The info structure is passed as the second argument. A static GHashTable is created and filled with dummy strings. The name of the class is entered as the key for the class pointer. This hash table is only used within the object class and not outside the file, object.c (i.e., static). The return value from *gts_graph_class* is a new class pointer if it doesn't already exist. Extra calls after initialization have no effect.

After all required parent classes have been created/initialized by the GTS_OBJECT_CLASS macro, a new GtsGraph object is created by the call to *gts_object_new*, which allocates memory for the structure and initializes it by calling *gts_object_init*. Quoting from internal documentation, function *gts_object_init*...

```
"Calls the init method of @klass with @object as argument. This is done
recursively in the correct order (from the base class to the top). You
should rarely need this function as it is called automatically by the
constructor for each class."
```

The init method is user supplied for GFS and part of the GTS library for its classes. For the GtsGraphClass, the init function (*graph_class_init*) is part of the GTS library. It initializes the write (*graph_write*) and read (*graph_read*) functions. These are referenced as "klass->read" in the code. The GtsObject pointer returned by *gts_object_new* is then passed to the macro GTS_GRAPH, which is replaced by the macro, GTS_OBJECT_CAST (GtsObject *, GtsGraph, *gts_graph_class*()). This checks the class and creates a GtsGraph object. The reason for this detailed class checking, creation, and initializing is the pseudo-object oriented structure being reproduced by GTS and GFS. It is necessary to have the creation of objects be independent of the program sequence.

The result of the line above is instantiations of the following structures within the gts_graph_read function, as well as all parents.

```
GtsGraphClass:
  GtsHashContainerClass  parent_class
  gfloat                 (* weight) (GtsGraph *)
```

where the argument to the function, *weight*, is a GtsGraph pointer.

The (GHashTable *items) are pointers to any data associated with the GtsGraph. The function *gts_hash_container_class* in-turn initializes a gts_container_class and assigns the *hash_container_add*, *_*remove*, *_*foreach*, and *_*size function*s. These are part of the GTS library. They use the glib c commands for hash containers (e.g., *g_hash_table_insert*).

Function *gts_graph_read* allocates the memory for (GtsGNode ** nodes), which is an array of pointers to GtsSListContainers. A new (GtsObject *new_node) is created for each of the nodes that is read from the **bath.gts** file. This node is filled by calling *gts_container_add*, which then calls the "add" function for the GtsContainerClass. The "add" function is *container_add*, which

is a short wrapper for calling the "add_container" function of the GtsContaineeClass. This member is initialized NULL in *containee_class_init* (line 27 of GTS/container.c).

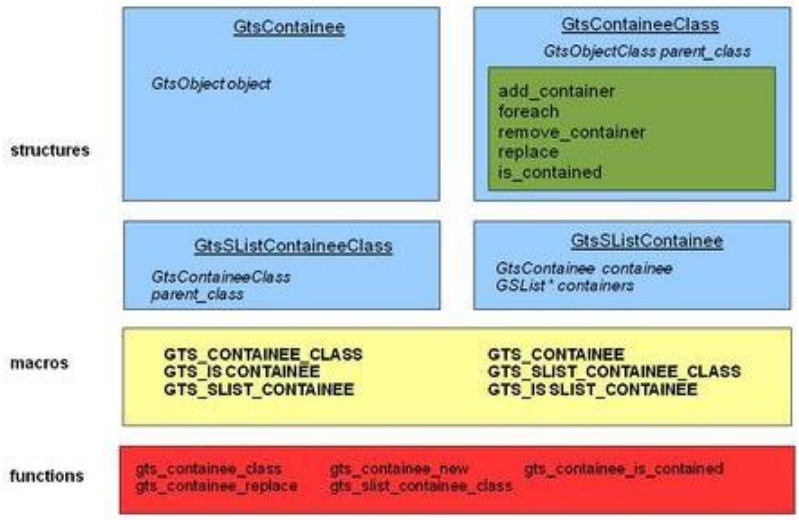
When *gts_graph_read* calls *gts_container_add*, it passes (GtsGraph *g) as arg[1] and (GtsObject *new_node) as arg[2]. Function *gts_container_add* then passes the arguments directly to the "add" function, which is *container_add*. To reiterate, the GtsGraph is the container and the new node is the item to be added (containee). Note that the new node is only an empty container. This *container_add* function then passes the node as arg[1] and the graph as arg[2] to the "add_container" function member, *slist_containee_add_container*. This function is a wrapper for *g_slist_prepend*, the glib function to prepend an item to a singly linked list. The new node from *gts_graph_read* is held in a temporary variable (GtsSListContainee *item). The (GSList *containers) member of this GtsSListContainee structure is searched for the GtsGraph structure, which has a (GtsHashContainer object) member. As expected, the GtsHashContainer includes a GtsContainer and a GHashTable. If the new node does not already have a pointer to the graph, its location (pointer) is prepended to the (GSList * containers) member of the GtsSListContainee that was searched. The containees pointer is updated to reflect this before returning.

Now that the new node has been added to the node structure, the values of the vertices can be read from file ***.gts** by "(g->node_class)->read". The node_class is a GtsGNodeClass* member of a GtsGraph. The "read" function for the GtsGNodeClass indicated on line 1457 of file graph.c is probably pointing to the GtsObjectClass "read" function through inheritance.

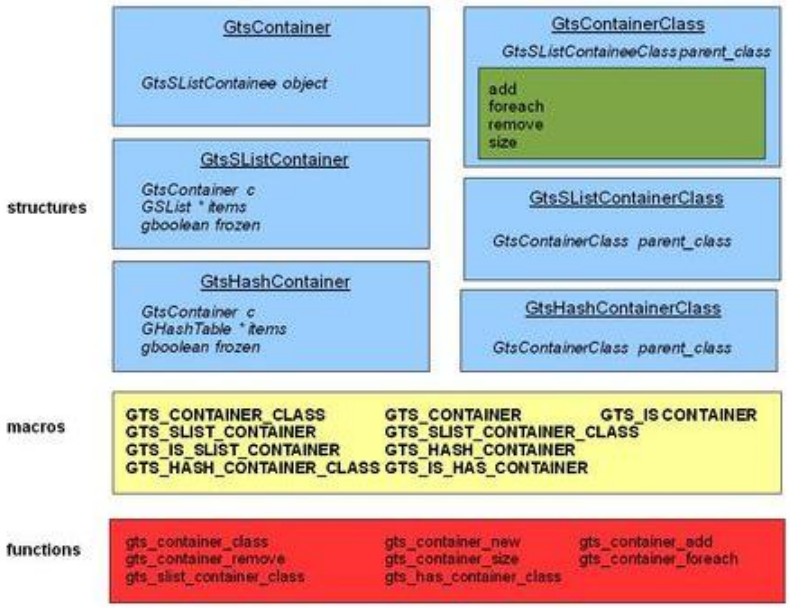
GtsContainers and GtsContainees

A containee (Figure 4A) is contained within a container (Figure B.8B), but the relationship between classes in GTS is more complex than this because a container can become a containee for another object. This semantics allows composite objects that are recursively contained within objects while also containing other objects.

Figure B.8. Class diagrams for containers.



A. Diagram for the GtsContainee Class.



B. Diagram for the GtsContainer Class.

For example, a GtsContainer has a GtsSListContainee object within it.

```
struct _GtsContainer {
  GtsSListContainee object;
};
```

This structure has only one member, which introduces a singly linked list object. A singly linked list simply lists items in a specified sequence for processing. The definition of this object shows how the linkage is accomplished using a singly linked list (i.e., GSList):

```
struct _GtsSListContainee {
  GtsContainee containee;
  GSList * containers;
};
```

This brings us to the basic GTS class, the GtsObject (Figure 3):

```
struct _GtsContainee {
  GtsObject object;
};
```

However, the GSList member (**containers*) is the key; this is part of the glib library. It has two members, a gpointer (generic pointer type) named *data*, and a GSList object named, appropriately enough, *next*. The surface vertices that were read from the ***.gts** file are stored in the *data* member whereas the next index to be processed from the list is held in *next*. This entire

complex hierarchy has been worked out for the GfsSolid object into which the water depth data are placed:

```
GtsSListContainer *                     solids
  GtsContainer                          c
    GtsSListContainee                   object
      GtsContainee                      containee
        GtsObject                       object
          GtsObjectClass *              klass
          GtsObjectClassInfo            info
            gchar                       name
            guint                       object_size
            guint                       class_size
            GtsObjectClassInitFunc      class_init_func
              void   (*GtsObjectClassInitFunc) (GtsObjectClass * objclass)
            GtsObjectInitFunc           object_init_func
              void                  (*GtsObjectInitFunc) (GtsObject * obj)
            GtsArgSetFunc               arg_set_func
              void                      (*GtsArgSetFunc) (GtsObject * obj)
            GtsArgGetFunc               arg_get_func
              void                      (*GtsArgGetFunc) (GtsObject * obj)
          GtsObjectClass *              parent_class
          void                          (* clone)
          void                          (* destroy)
          void                          (* read)
          void                          (* write)
          GtsColor                      (* color)
            gfloat                      r
            gfloat                      g
            gfloat                      b
          void                          (* attributes)
        gpointer                        reserved
        guint32                         flags
      GSList *                          containers
        gpointer                        data
        GSList *                        next
          Repeated recursively as required...
    GSList *                            items
      gpointer                          data
      GSList *                          next
    gboolean                            frozen
```

A GtsContainer can be a member of a range of structures because its sole purpose is to supply a linked list with a hash table for accessing the members of the list in a specified manner. I printed the pointer to the "add_container" member for the GtsContaineeClass in function *container_add*. I also printed the pointer to function *slist_containee_add_container*, which is the "add_container" member for the GtsSListContaineeClass. They are the same. This is because the GtsContaineeClass is the parent of the GtsSListContaineeClass. This is the function that adds items to a container. Function *slist_containee_add_container* is a wrapper for *g_slist_prepend*. The first argument is a (GtsContainee *) and the second is a (GtsContainer *).

GtsHashTables

It convenient to define a GtsHashContainer as follows:

```
struct _GtsHashContainer {
  GtsContainer c;
  GHashTable * items;
  gboolean frozen;
};
```

The GtsHashContainer structure is a wrapper for a GHashTable, which is part of the Glib library that comes with the operating system. It also has a GtsContainer member that inherits the functionality of this structure (Figure B.4B). The *items* member points to a hash table. This structure (*aka* class) also contains a GtsContainer, which is a collection of other objects. In the case of the GTS library, this collection is contained in a GtsSListContainee object.

A GtsHashContainerClass (GtsHashContaineeClass) structure is a type of GtsContainer/GtsContainee (Figure B.2).

```
struct _GtsHashContainerClass {
  GtsContainerClass parent_class;
};
```

Glib functions are used to access data contained within (GtsSurface *)->(GHashTable *faces). These functions are discussed in the glib documentation. Typical Glib functions used to access data in a GHashTable are:

```
void      g_hash_table_insert  \
              (GHashTable *hash_table, gpointer key, gpointer value);
gpointer  g_hash_table_lookup  \
              (GHashTable *hash_table, gconstpointer key);
void      g_hash_table_foreach \
              (GHashTable *hash_table, GHFunc func, gpointer user_data);
```